BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

As rescanning documents *will not* correct images, please do not report the images to the Image Problem Mailbox.

PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 5 : G06F 15/16 | A1

(11) International Publication Number: WO 93/14459

(43) International Publication Date: 22 July 1993 (22.07.93

(21) International Application Number: PCT/US93/00365

(22) International Filing Date: 19 January 1993 (19.01.93)

(30) Priority data:
07/827,030 17 January 1992 (17.01.92) US

(71) Applicant *(for all designated States except US)*: CAELUM RESEARCH CORPORATION [US/US]; 11229 Lockwood Drive, Silver Spring, MD 20901 (US).

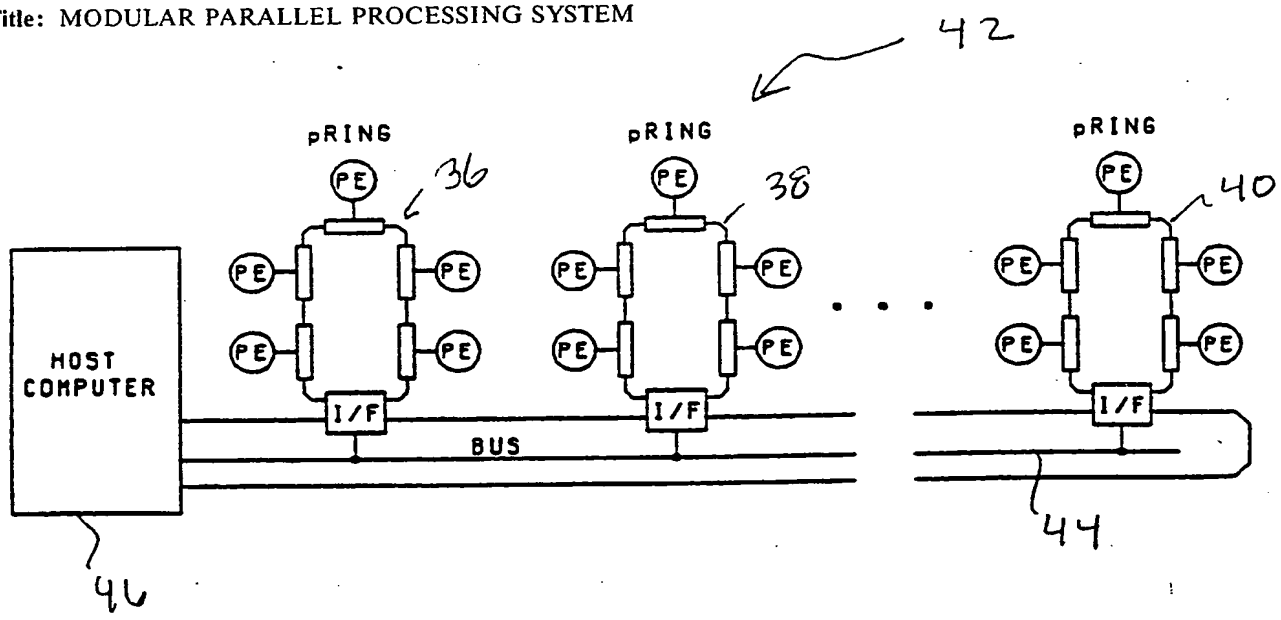
(72) Inventors; and
(75) Inventors/Applicants *(for US only)* : CHIOU, Yu-Shu [CN/US]; 19902 Halfpenny Place, Gaithersburg, MD 20879 (US). JUMP, Lance, B. [US/US]; 14401 Sturtovant Road, Silver Spring, MD 20904 (US). LIGOMENIDES, Panos, A. [US/US]; 8802 Magnolia Drive, Lanham, MD 20706 (US).

(74) Agent: HASHIM, Paul, C.; Venable, Baetjer, Howard & Civiletti, 1201 New York Avenue, N.W., Washington DC 20005 (US).

(81) Designated States: CA, JP, KR, RU, US, European paten (AT, BE, CH, DE, DK, ES, FR, GB, GR, IE, IT, LU MC, NL, PT, SE).

Published
With international search report.

(54) Title: MODULAR PARALLEL PROCESSING SYSTEM



(57) Abstract

A modular neural ring (MNR) system is provided for neural network processing which comprises one or more primitive rings (36, 38 and 40) embedded in a global communication structure (GCS) (44). The MNR bus (44) is a multiple access, multi-master, arbitration, hand-shaked data bus. Each primitive ring is a single instruction stream, multiple data stream (SMD) machine being a control unit for controlling in parallel a number of processing elements (PEs) (54, 56, 58) connected by a local communication network (64). Within a primitive ring, a master controller (86) controls housekeeping functions, scratch pad memory and synchronization. A processor controller (88) transmits signals to all of the PEs on the primitive ring to carry out vector processing. An interface controller (90) controls the primitive ring's external interfaces to the GCS (44). Computation within a processing element is performed by processor logic blocks (PLB) (150). Each PLB implements a RAM based shift register scheme.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AT	Austria
AU	Australia
BB	Barbados
BE	Belgium
BF	Burkina Faso
BG	Bulgaria
BJ	Benin
BR	Brazil
CA	Canada
CF	Central African Republic
CG	Congo
CH	Switzerland
CI	Côte d'Ivoire
CM	Cameroon
CS	Czechoslovakia
CZ	Czech Republic
DE	Germany
DK	Denmark
ES	Spain
FI	Finland
FR	France
GA	Gabon
GB	United Kingdom
GN	Guinea
GR	Greece
HU	Hungary
IE	Ireland
IT	Italy
JP	Japan
KP	Democratic People's Republic of Korea
KR	Republic of Korea
KZ	Kazakhstan
LI	Liechtenstein
LK	Sri Lanka
LU	Luxembourg
MC	Monaco
MG	Madagascar
ML	Mali
MN	Mongolia
MR	Mauritania
MW	Malawi
NL	Netherlands
NO	Norway
NZ	New Zealand
PL	Poland
PT	Portugal
RO	Romania
RU	Russian Federation
SD	Sudan
SE	Sweden
SK	Slovak Republic
SN	Senegal
SU	Soviet Union
TD	Chad
TG	Togo
UA	Ukraine
US	United States of America
VN	Viet Nam

TITLE OF THE INVENTION:

MODULAR PARALLEL PROCESSING SYSTEM

BACKGROUND OF THE INVENTION:

The architecture and nature of neural network processing machines promise a solution to certain kinds of problems, that are too slow if not impossible to solve even on most powerful currently available computer. When research interest expanded to artificial intelligence, researchers realized that only limited progress can be made with current computing technologies. Computers are currently limited by their serial Von Neumann-type of architecture, and because they are essentially discrete symbol-processing machines.

Human beings are not purely logical, nor can human behavior be regulated by mathematical or logical formulas. Human beings do not make decisions based on evaluating several hypotheses through formal probabilistic methodology, nor do human beings go step by step through any existing pattern-recognition algorithm to recognize objects. Observations of human behavior indicate that it is very difficult to achieve a state of discerning intelligence without an inductive processing tool.

According to the definition given by the Defense Advanced Research Projects Agency (DARPA), "The Neural Network is an information processing system which operates on inputs to extract information, and produces outputs corresponding to the extracted information ... Specifically, a neural network is a system composed of many simple processors-fully, locally, or sparsely connected-whose function is determined by their interconnection topology and strengths. The system is capable of a high-level function, such as adaptation or learning with or without supervision, as well as lower-level functions, such as vision and speech pre-processing. The function of the simple processor and the structure of the connections are inspired by biological nervous systems".

The key attributes of neural network functions are massive parallelism and adaptivity. The massive parallelism results in high-speed performance and in potential fault tolerance. Adaptivity means that the neural networks can be trained rather than programmed, and their performance may improve with experience. Another important advantage of neural networks is that they enjoy parallel processing while remaining simple to use. From an information

processing viewpoint, neural networks are pattern-processing machines whose function is akin of the inductive inference associated with human brain functions. This is unlike present available, deductive computers which are based upon symbolic logic processing.

Figure 1 illustrates the type of biological neuron (20) which has influenced the development of artificial neural networks. The synapse (22) is the tissue connecting neurons. It is capable of changing a dendrite's local (24) potential strength in a positive or negative direction, depending on the pulse it transmits. These transmissions occur in very large numbers, but since they are chemical, they occur fairly slowly. The neuron is the processing element in the brain, that is, it receives input, performs a function and produces an output. The neuron has two states: firing or not firing. The synapse is basically composed of the data-line connecting neurons, but is much more. A synapse may be inhibitory or excitatory. When a neuron's output is connected through an inhibitory synapse to another neuron, the firing of that neuron discourages the firing of the neuron to which the signal goes. On the other hand, a neuron receiving an input through an excitory synapse will be encouraged to fire. Synapses may also have weights associated with them, which indicate the strength of the connection between two neurons. If the firing of one neuron has a lot of influence on the firing of another, the weight of the synapse connecting them will be strong. The human cerebral cortex is comprised of approximately 100 billion (10^{11}) neurons with each having roughly 1,000 dendrites that form some 100,000 billion (10^{14}) synapses. The system functions at 10,000 billion (10^{16}) interconnections per second if it operates at about 100 Hz. The brain weighs approximately three pounds, covers about 0.15 square meters, and is about two millimeters thick. This capability is absolutely beyond anything that can be presently constructed or modeled. Understanding how the brain performs information processing can lead to a brain-like model, and possible implementation in hardware.

Artificial neural networks (ANNs) are inspired by the architecture of biological nervous systems, which use many simple processing elements operating in parallel to obtain high information processing rates. By copying some of the basic features of the brain into a model, the ANN models have been developed which imitate some of the abilities of the brain, such as associative recall and recognition. ANNs are general-purpose pattern processing machines, but there are specific classes of problems for which various types of

ANNs are best suited. Being parallel in nature, neural networks are best suited for processing intrinsically parallel data-processing tasks. Thus, they are good for problems such as image-processing and pattern-recognition, vision and speech-processing, associative recall, etc. The characteristics that artificial neural networks hope to provide are:

1. Tolerance to removal of a small number of processing elements,
2. Insensitivity to variations between processing elements,
3. Primarily local connectivity and local learning rules,
4. Real time response, and
5. Parallelism.

There are two major architectural approaches to the implementation of large scale ANNs: (i) using a very high speed central processor; and (ii) implementing a fully parallel processing system. The former is the typical ANN simulation on general purpose computers and on neurocomputers. The latter is usually found in small-scale, special purpose devices. Both of the architectural approaches suffer from most of the following limitations: As the size of a neural network grows by a factor of N, the interconnections grow by a factor of N^2. For fully parallel implementation, the capacity becomes an upper limit. For serial central processing, the speed decreases as N increases. Some of the electronic virtual neuroncomputers implemented on a parallel architecture are: the Connection Machine, Warp, AAP-2, and Transputer. However, these architectures have not solved efficiently the connectivity problem, as needed for simulation of large-scale neural networks, which prevents their expendability. Since these are multiple instruction-stream multiple data-stream (MIMD) systems, controller operation expense tends to be disproportionately great with respect to processing power. In other words, great expense contributes little to processing power.

A neuron in an ANN system is capable of accepting inputs from many neurons, and capable of broadcasting its activation value to many other neurons in the system through weighted interconnections. The ability to "memorize" and to "learn" in a neural network system is derived from the weighted interconnections. Every neuron should contribute its activation value to the state of the system. For an N neuron system, the potential fan-in and fan-out requirement is N^2-1 in a fully connected model. This requirement increases

exponentially with the number of neurons. The fan in/out problem limits the fully parallel implementation of ANNs using VLSI technology. Present solutions to massive connectivity and fan in/out problems associated with interconnecting enormous numbers of processing elements with 2-D layouts have been unsatisfactory in producing the collective computational performance predicted by the theoretical models.

There are many different scenarios in which neural networks can be implemented and each of them has its own speed requirements. Furthermore, speed requirements are determined by the tasks being performed. However, the fully parallel architecture is most advantageous if the number of neurons is limited. Despite advantages provided by the superior speeds of electronic processors, simulation of large-scale neural networks on serial computers suffers from low throughput. The simulation throughput of the serial general purpose CPU falls rapidly when the size of the neural network grows. The environment in which a neural network simulation is placed is a major determinant of its speed requirements. The most stringent requirements to speed occur in real time applications of ANNs. However, there are presently only a few systems which can meet the challenge at the present time.

Utilization is the ratio of hardware in operation at any instant to the overall hardware of the system. According to this definition, a general purpose Von Neumann processor suffers from low utilization in simulating neural networks. The processing elements of neural networks execute multiply-accumulate instructions most of the time. Thus approximately 80% of the general purpose processor is idling during the simulation of neural networks. Though existing multiprocessor system could employ higher parallelism required by ANNs, their hardware utilization would be limited due to the communications bottleneck between processors. Those multiprocessor systems have expensive general purpose processors. The waste of processor resources is a severe problem from an economic point of view. The fully parallel architecture suffers from low hardware utilization when applying partially connected ANN model to the architecture, or when applying a smaller model than its hardware-configuration calls for.

ANN models have been suggested for pattern-recognition, associated memory, optimization, among other functions. An ideal ANN simulator should be reconfigurable, so that it may realize the various existing models, as also the ANN models yet to come.

Existing ANN simulators are either model-dependent or limited to selective models. As the field progresses, new ANN models are constantly being developed. Researchers need an implementation tool which would allow modular and reconfigurable realization of ANNs, in order to help develop their ideas. Current ANN simulators are dedicated to those ANN models they intend to simulate. One would need to redesign the system to meet one's special needs. What is needed is a reconfigurable and modular architecture implementation of ANNs, so that various topologies and size of ANNs may be realized efficiently.

What is needed for both the theoretical development and the commercialization of ANNs is large-scale implementation architectures and technologies. Modular, massively parallel architectures are the most promising in terms of scaling and extending neural system capabilities. But the overwhelming problem associated with massive parallelism is efficient communication. Massively parallel architecture machines such as Connection Machine, NCUBE, and transputer, which are based on a hypercube communications topology, perform quite well on certain models with local connectivity. However, the communication structure of these machines can lead to dramatic decreases in performance for more general models. Also, there are substantial portions of the hardware costs of these machines which are dedicated to control units.

What is desirable in neural network implementation is a universal modular architecture, which will allow VLSI hardware implementation of large scale neural networks. There are several requirements to the architecture:

1. be highly parallel,
2. allow for high hardware utilization,
3. solve the network communications problem without fan in/out limitations,
4. be cost-effective in meeting performance goals with speed versus hardware complexity trade-off,
5. be modular with easy interconnection facilities for the design of large systems with varying requirements of connectivity and configuration,
6. be expandable with no added communication problems, while maintaining connectivity, to allow operation at raised levels of complexity and neural volumes,

7. be endowed with switchable connectivity so as to allow for dynamically reconfigurable (by software) neural network system architectures, which may implement different theoretical neural network models without major redesigning or reconstruction, and be implementable with state of the art fabrication techniques.

As stated previously, it is hoped that biological neural network computation principles can be applied to artificial neural networks (ANNs) to help solve difficult problems of recognition, association, optimization and other combinatorially complex problems. But massively parallel computers have been developed and have still not attained human performance in many difficult problem areas. Two key obstacles facing parallel computation are problem decomposition (and representation) and communication. In order to use massive numbers of processing elements on a problem, the problem must be parallelized and mapped onto the processing elements in such a way that dependencies do not cause much of the hardware to remain idle. Neural processing can be viewed as such a decomposition and, if an understanding can be developed to allow difficult problems to be solved using neural techniques, then one of the obstacles will be overcome. If a problem can be cast in neural network terms, then we have a parallel decomposition of the problem.

But there still remains the communication problem. The difficulty with neural models is that vast amounts of information must be communicated among neurons in the system. Three important characteristics of biological neural networks make this possible. First, and perhaps most important, is the three-dimensional connectivity structure of biological neural networks contrasted with the planar connectivity for integrated electronics. Second, the processing is much more distributed than suggested by the large number of neurons available. In biological neural networks, each synapse (corresponding to a weight in the ANN) is a processor which performs a multiplication as well as acts as a storage element. Accumulation of a weighted sum is performed along the dendrites thus making them also large distributed processors (accumulators). In fact, since the processing is distributed and largely carried out along the pathways between neurons (axons, synapses and dendrites), much of what might be considered communication hardware is actually computation hardware. Finally, because of the long processing times in each neuron, the data rates are relatively low thereby allowing

less communication hardware. In fact, there is evidence that communication delays are actually of computational importance in neural algorithms. Pulse coded activations and regenerative conduction pathways contribute to robust operation in the presence of noise as well as allowing long distance connectivity.

Most of the processing time used in an ANN is for accumulation of the weighted sum, at least in the operation (also called retrieval) phase. The adaptation or learning phase also requires computation, as will be discussed later. For a fully connected network of N neurons, N^2 multiply-accumulates are required to compute all weighted sums for one network cycle. The amount of computation involved in computing the activation function varies depending on its complexity, but it scales linearly with N since the activation function is applied pointwise to the sum vector. Furthermore, most of the hardware in an ANN system is devoted to storing the weight matrix. For these reasons, ANN hardware speed is measured in interconnections per second, and capacity is measured in total number of interconnects that can be stored. In a 1988 DARPA neural network study, heavy use was made of a performance plane with these two dimensions. These measures do not give a complete picture, particularly for general purpose ANN hardware which is not designed for a specific application. Additional characteristics which are desired, and found in the proposed architecture are summarized below.

A general purpose implementation architecture must support a wide variety of models. This includes support of various connection topologies, activation functions and learning paradigms. The degree to which a particular implementation architecture can support a variety of models is its flexibility. However, in many environments, this flexibility is preferably carried one step further to programmability and dynamic reconfigurability. For example, restucturable VLSI meets the criteria of flexibility, but the reconfiguration steps are one-way, one-time static restructuring. An ANN workstation, for example, is preferably able to be programmably reconfigured to suit the model under investigation.

The ability of the architecture to be scaled (in some set of dimensions) is its extensibility. CPU based architectures are extensible in the size dimension but do not scale well along the speed dimension. Modular extensibility allows the system to be scaled with the addition of well defined modules and interfaces. Clearly, it is desirable to have modular

extensibility for large scale ANN implementation architectures. Ideally, modular extensibility would include field upgradeable expansion of existing systems by the addition of standard modules.

Although speed, capacity, flexibility and modular extensibility are important, desirable properties, they all incur some costs. These costs are ideally minimized in a good implementation architecture. Thus, efficiency is an important property in an implementation. Efficiency is viewed both in terms of implementation efficiency as well as operational utilization.

An architecture should allow trade offs to be made between the properties listed above. Trade offs are possible in three major regimes. First, the system designer can make trade-offs during the design and construction of a particular instance of the system. Second, the modular field upgrades can be made again making performance-cost trade-offs. Finally, ANN applications can trade various parameters such as precision with speed during specific model implementation.

Processing in ANNs can be divided into two distinct phases: the application phase, and the adaptation phase. The application phase is commonly referred to as the retrieval phase in the literature, and this terminology originates from the use of ANNs as associative memories. The application phase is the most consistent among ANN models. The processing for each neuron in the application phase for the bulk of the models reported can be represented by an activation function applied to a weighted sum of the inputs to that neuron. Although some early models used linear activation functions, modern networks invariably employ nonlinear functions such as steps, sigmoids and linear thresholds. Thus, the processing for neuron i can be represented by

$$v(t+1)_i = f(_i \sum_{j \in J} w_{ij} v(t)_j)$$

Where J is the set of neuron outputs that feed neuron . Taken as a whole, the application phase processing can be represented by a matrix-vector multiply followed by a pointwise vector activation function.

$$V(t+1)=F(WV(t))$$

The last equation describes a discrete time, synchronous system where all neuron outputs are assumed to be updated at the same time step. Also, the weights are assumed not to be functions of time during the application phase. The weights may vary with time if the network is adaptive, but since this variation is performed in a separate phase, and because the weights variations are usually slow, it is appropriate to represent the weights as fixed in the application phase. The components of the vector function are, in general, different from one another.

Not all ANN models perform such uniform processing of all inputs to a neuron. For example, some of the neurons in ART treat inhibitory inputs differently from excitatory inputs. Other models, such as the Neocognitron developed by Fukushima, combine sums from different clusters of input neurons. This can be accommodated in the above formulation by defining subneurons of each more complex neuron and then combining their outputs. The sums proceed as indicated above, i.e., the combination is performed in the pointwise activation function which is now a function of several inputs. Alternatively, but equivalently, inputs to a neuron can be classified and the activation function applied to the class sums.

The adaptation phase varies considerably among ANN models. It is difficult to express the general adaptation phase processing as succinctly as the application phase. However, the following equations can represent the adaptation processing for most of the mainstream ANN models.

$$w_{ij}(t+1)=w_{ij}(t)+\Delta w_{ij}(t)$$

Determination of the Δw varies considerably among ANNs, but the following functional form is sufficiently general to include most ANN models.

$$\Delta w^{l}{}_{ij}=G(w^{l}{}_{ij},h^{l-1}{}_{j},S^{l}{}_{i},S^{i+1}{}_{j},T)$$

In this equation, l indexes the layer which contains the neuron receiving stimulation

via the weight. The h terms are some local function of the state of the corresponding neuron. This is usually that neuron's output or accumulator value. In some cases (such as with supervised learning like Backpropagation) h may include target pattern information local to an output neuron. The S terms are summary terms for the corresponding layer. The generalized delta learning rule uses this to back propagate error gradient information. These can also be used in reinforcement learning as nonspecific response grading inputs. The role of T is the environmental or critic's input in reinforcement learning. Competitive learning neighborhoods can be established by incorporating winner information in S. Learning rate, momentum, plasticity decay and other such parameters are incorporated in the overall function G.

The above formulation is sufficiently general to cover a wide variety of ANN models. These include Perceptron learning, Widrow Hoff, Backpropagation, Hopfield's outer product construction, the linear pattern associator, Kohonen's self organizing feature maps, most Hebbian and modified Hebbian learning, Oja's principal component extractor, vector quantization and adaptive resonance networks. Because the architecture is programmable with a fairly capable instruction set and much flexibility, it is possible to implement algorithms that are not represented by existing equations. Since the ANN models are constantly being introduced and improved, this flexibility is essential.

SUMMARY OF THE INVENTION:

A Modular Neural Ring (MNR) architecture is described below in accordance with the present invention. The MNR architecture is a collection of primitive processing rings (p rings) embedded in a global communication structure, which realizes the above described requirements desired in a large scale implementation architecture for an ANN.

The essence of the MNR architecture is a collection of primitive processing rings (pRings) embedded in a global communication structure. The pRings are SIMD machines with a control unit parallel serving a large number of attached processing elements (PEs). The PE's within a pRing are connected by a local ring communication network. Each pRing executes its own control program which synchronously and parallely controls the attached PEs. However, each pRing is potentially executing a different control program, thus the

processing nature of the overall MNR system is MSIMD.

The specific system architecture which was prototype is a bussed pRing supported by a host computer. Each pRing makes a connection to the system bus and to it's left and right neighbors. The connections to adjacent pRings allow for logically grouping a number of pRings to form a larger processing ring (called a slab). The bus is provided for more arbitrary communication between slabs.

The major computation in an ANN is the matrix-vector multiply. But aside from comprising the bulk of the processing, this operation also comprises the greater part of the communication requirements. For now, attention will be focussed on a fully connected network (e.g. a Hopfield net), but more general connectivity will be discussed in the next section. In this type of network, each neuron (or PE) requires the activation level of all other neurons in order to compute its weighted sum. If one neuron is assigned to each PE then N multiply-accumulates are required in each PE to complete the weighted sum phase of the processing. Each weighted sum is computed sequentially within the corresponding PE and thus each PE requires only the activation level of one neuron at a time. If the processing is properly phased within each PE then the activation levels can be placed on a ring and circulated such that they arrive at each PE at just the right time.

DETAILED DESCRIPTION OF THE DRAWINGS:

These and other features and advantages of the present invention will be more readily apprehended from the detailed description when read in connection with the appended drawings, in which:

Fig. 1 illustrates a biological neuron;

Fig. 2 is a schematic diagram of a modular neural ring (MNR) architecture;

Fig. 3 is a schematic diagram of a modular neural ring (MNR) architecture configured as a single in accordance with the present invention;

Fig. 4 illustrates a bussed pRing architecture constructed in accordance with the present invention;

Fig. 5 is a schematic diagram of a primitive ring (pRing) of processing elements (PEs) constructed in accordance with the present invention;

Fig. 6 is a schematic diagram of three controllers provided with a pRing constructed in accordance with the present invention, namely a master control unit (MCU), an interface control unit (ICU) and a PE control unit (PCU);

Fig. 7 is a schematic diagram of an MCU constructed in accordance with the present invention;

Fig. 8 illustrates a programmer's model of a pRing;

Fig. 9 is a graph illustrating speed and capacity characteristics of analog, fully parallel architectures, serial central processor architectures and the pRing architecture of the present invention;

Fig. 10 is a schematic diagram of an artificial neural network (ANN) workstation constructed in accordance with the present invention;

Fig. 11 is a schematic representation of a virtual ring composed of several pRings in accordance with the present invention;

Fig. 12 is a schematic diagram of a pRing constructed in accordance with the present invention;

Fig. 13 is a schematic diagram of a PE string board constructed in accordance with the present invention;

Fig. 14 is a schematic diagram of a processor logic block (PLB) provided within a PE string board in accordance with the present invention;

Fig. 15 is a schematic diagram of a shift register simulation scheme constructed in accordance with the present invention;

Fig. 16 is a block diagram of a PCU constructed in accordance with the present invention;

Fig. 17 is a block diagram of an MCU constructed in accordance with the present invention;

Fig. 18 is a block diagram of an ICU constructed in accordance with the present invention;

Fig. 19 illustrates bus transmission timing on an MNR bus constructed in accordance with the present invention;

Fig. 20 is a state diagram of bus transmission protocol in accordance with the present invention;

Fig. 21 illustrates the data processing hierarchy at which the MNR architecture of the present invention is programmed;

Fig. 22 is a schematic diagram of the MNR language hierarchy in accordance with the present invention;

Fig. 23 is a schematic diagram of subprocesses relationships associated with SimM, a simulation tool developed in accordance with the present invention for simulation of the MNR architecture;

Fig. 24 illustrates a construction phase of SimM;

Fig. 25 illustrates a simulation phase of SimM;

Fig. 26 illustrates global control and program loader modules of SimM developed in accordance with the present invention.

Fig. 27 illustrates a pRing module developed for use with SimM;

Fig. 28 illustrates a HOST module developed for use with SimM;

Fig. 29 illustrates a global communication control module developed for use with SimM;

Fig. 30 illustrates a monitor module developed for use with SimM;

Fig. 31 is a graph illustrating the performance with MNR architecture of the present invention on a DARPA;

Fig. 32 is a graph illustrating the effects of speed versus the number PEs on the MNR architecture of the present invention;

Fig. 33 is a graph illustrating the effects of speed versus the neuron PE ratio on the MNR architecture of the present invention;

Fig. 34 is a graph illustrating the effects of speed versus the number PEs on the MNR architecture of the present invention;

Fig. 35 is a graph illustrating the effects of speed versus the neuron PE ration on the MNR architecture of the present invention;

Fig. 36 is a graph illustrating the effects of speed versus pRing size on the MNR architecture of the present invention;

Fig. 37 is a graph illustrating PCU utilization versus pRing size of the MNR architecture. Fig. 38 is a graph illustrating ICU utilization versus pRing size of the MNR architecture;

Fig. 39 is a graph illustrating speed versus communication bandwidth of the MNR architecture;

Fig. 40 is a graph illustrating PCU utilization versus communication bandwidth of the MNR

architecture;

Fig. 41 is a graph illustrating ICU utilization versus communication bandwidth of the MNR architecture;

Fig. 42 and 43 are graphs comparing device utilization of the PCU and ICU;

Fig. 44 is a graph illustrating speed versus communication bandwidth of the MNR architecture;

Fig. 45 is a graph illustrating PCU utilization in the MNR architecture of the preset invention;

Fig. 46 is a graph illustrating ICU utilization in the MNR architecture of the preset invention;

Fig. 47 is a graph illustrating speed versus the precision of the MNR architecture of the present invention;

Fig. 48 is a graph illustrating PCU utilization versus the precision of the MNR architecture of the present invention;

Fig. 49 is a graph illustrating ICU utilization versus the precision of the MNR architecture of the present invention;

Fig. 50 is a graph illustrating performance characteristic of the MNR architecture of the present invention;

Fig. 51 is a graph illustrating cost and performance estimates of the MNR architecture of the present invention;

Fig. 52 is a graph is a graph illustrating the performance of a two pRing MNR prototype;

Fig. 53 is a graph illustrating the performance of a forty pRing MNR prototype;

Fig. 54 is a graph illustrating the performance of a multi-layered feed forward MNR prototype;

Fig. 55 is a graph illustrating the performance of an error back propagation (BP) MNR prototype; and

Fig. 56 is a graph illustrating the PE utilization of BP utilization of BP implementation in an MNR prototype.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS: PRINCIPLES OF OPERATION OF THE MODULAR NEURAL RING (MNR) ARCHITECTURE

The basic processing ring of the present invention is configured as a synchronous ring communication network with K processing elements (PEs) situated radially off the ring as shown in Figure 2. Although the ring is the basic theme, variations are presented to suit different ANN topologies. The simplest case to describe is a fully connected N neuron network assigned to a ring of N PEs (K=N). It is also a simple matter to have K < N, thereby allowing one PE to serve several virtual neurons. The PEs 30 operate synchronously and in parallel on data delivered to them by a data vector, which circulates in the ring communications network 20.

The architecture of the neural ring is basically a Single Instruction-stream Multiple Data-stream (SIMD) processing structure with sequenced delivery of identical data sets to the PE 30, rather than partially processed data through each processor, as may happen in pipelined processing of systolic architectures. The operations of the neural ring are highly parallel, allowing for very high processing element utilization due to timely delivery of data to the PEs. The ring communication structure allows for simple interconnection and extensibility schemes, without major layout problems stemming from fan in/out or connectivity requirements in VLSI implementations. The regularity of the neural network topology also allows efficient replication of the rudimentary PEs, which are served in clusters by the more complex control unit.

To understand the operation of the neural ring, consider the case of a fully connected neural network with K PEs and N neurons, where $K=N$. A network cycle is completed when the activation outputs of all N neurons have been updated. The cycle requires that the weighted sum for every neuron's input in the systems is accumulated, i.e.n N^2 multiplications and N^2 additions. These operations are carried out concurrently by the K PEs, where each PE performs a multiplication and an addition or accumulation (MAC) in one primitive computation time.

At the start of a network cycle, each PE places its neuron's value on the ring and performs a MAC operation. During the MAC, the data are rotated clockwise on the ring-

an operation that is designed to take the same time as the MAC. Each PE then performs a MAC with the new value on its ring input while the activation vector moves another step on the ring. When the activation vector has made a complete tour of the network, all weighted sums have been accumulated and the activation function is applied in each PE. The system is now ready for another network cycle. Note that under the assumptions that the ring step time is equal to the processor MAC time there is no processor idle time. Synchronously, the computed sums are mapped through the nonlinear activation functions of each neuron (e.g., hardlimiter, sigmoid, etc.) to produce the neuron outputs, and the next network cycle begins. Since the data-circulation time is overlapped with the primitive computation time, there is no PE idle time within the network cycle.

Each PE is preferably supported by a weight memory, which represents a horizontal slice of the weight-matrix, and by an accumulator-memory which allows each physical PE to serve multiple neurons. This provides a ready mechanism for cost versus speed trade-offs. It is also flexible enough to accommodate various adaptation algorithms since each PE has access to a slice of the weight-matrix. Different activation functions can be achieved using polynomial approximations. State-dependent behavior can be implemented using the associated accumulator-memory to store state information. The pointwise functions and matrix-multiplication required for the learning operation can also be carried out in a similar manner, although the logical connection topology for the learning phase of the network operation is often different from that of the application phase.

Although there is great regularity in the connectivity and in the processing performed by neurons in a network, for most models this regularity applies, for the most part, locally to groups of neurons. Hence, the ring communication structure is most suitable locally. A different global control and communication structure is usually required for the entire neural network system. If the architecture is to be used to implement a variety of models, then a dynamically reconfigurable global structure is needed.

The locality property of typical neural networks refers to the rule that if a neuron connects to another neuron in the network, it is likely also to be connected to that neuron's neighbors. Thus, if a neuron requires another neuron's value, it is likely to also require the values of the other neurons in its neighborhood. A *pRing* (primitive Ring) is an indivisible,

ring-configured collection of processing elements. The idea is to assign groups of logical neurons to pRings in such a way as to take advantage of local regularity, but still be able to accommodate global specialization. In accordance with the present invention, the collection of pRing characterizing the MNR architecture comprises a communication structure having a common bus and bidirectional connections between adjacent pRings. Figure 3 shows an MNR system of three consecutive pRings configured as a single neural ring. The connecting adjacent pRings allows for several consecutive pRings to be configured as one, large, virtual ring 42 which can represent, for example, a neural slab. The bus connection 44 allows for communication between pRings that are not adjacent. A host computer 46 is connected to the bus and serves as the data and command I/O processor.

The pRings 36, 38 and 40 are SIMD machines comprising a central control unit48, 50 and 52, respectively, which serves a large number of peripherally attached PEs operating in parallel. The PEs within a pRing are connected by a local ring-communication network. Each pRing executes its own control program, which synchronously controls in parallel the attached PEs. However, each pRing can potentially execute a different control program from other pRings, thus, making the processing nature of the overall MNR system MIMD.

An SIMS ring architecture is generally known. In accordance with the present invention, an SIMD ring is modified to be primitive to all for the employment of modular primitive rings (pRings) as *components* in an ANN system.

With *untried* reference to Figure 3, the processing for all PEs within a ring is preferably identical, with phased delivery of the data vector to all PEs in the ring. Such a system is a prime candidate for SIMD/systolic processing. Thus, each PE can be kept quite simple with the more complex control unit being used only once per ring. (In a general purpose CPU, more circuitry is dedicated to control than to computation.) VLSI implementation exploits the remarkable regularity of such a system to help minimize development costs. The architecture can be implemented with a central controller which broadcasts instructions to PE strings. PEs within a string are connected point-to-point in a line. The strings are concatenated and the end strings connected together to form a ring. With such an implementation, the system is modularly extensible and even field upgradable by inserting additional PE strings. The additional PEs place no additional communication

burden on the system since each brings along its own point-to-point connection to its neighboring PEs. Thus each PE requires only a pair of simplex communication links.

With reference to the single ring architectures, there are some shortcomings. When the virtual topology is other than fully connected, the efficiency of the system can deteriorate. The system performance can also degrade if K is not compatible (in a sense to be defined later) with N. If the goal is a general purpose architecture on which to run various ANN models (an ANN workstation for example) then the ring may not be the best structure.

In order to attain maximum PE utilization the processing load is preferably evenly distributed among the PEs. Also, the communication network should be able to deliver the data where and when it is needed. Since the ring is SIMD, then K must divide N. But, since this is unlikely for an arbitrary network, null neurons can be introduced to augment N. But the processing associated with the null neurons is equivalent to processor idle time since the model did not actually call for it. So an ANN model with 502 neurons implemented on a ring of 100 PEs will require 98 null neurons resulting in a real utilization of about 70%.

The communication network may also lead to a degraded utilization by PEs waiting to receive for data and thus making them idle. A locally connected network implemented on a unidirectional ring will still require an entire network cycle to send the data to every PE that requires it, however, for much of the time, the PEs will be receiving data that they cannot use. The solution, as suggested above, is to use a bidirectional ring, but each ANN topology will have its own requirement on the network and there are substantial costs associated with general connectivity schemes.

Slab oriented ANNs, such as the Neocognitron and the Back Propagation Net, can compound the utilization problem. For such systems implemented on a single SIMD ring, a time multiplexed assignment of slabs onto the ring is required. Processing will proceed in major phases with each major phase corresponding to a different slab. Now utilization losses due to null neuron augmentation will occur for each slab. Furthermore, the communication of data may be much more complex.

The general structure of a bussed-pRing architecture is shown in Fig. 4. The bussed-pRing architecture is flexible enough to support a wide spectrum of virtual topologies efficiently. Each pRing 36, 38 and 40 makes a connection to the system bus 44 and to its

left and right, adjacent pRings. The connections to adjacent pRings allow for logically grouping a number of pRings to form a larger processing ring (called a slab). The bus is provided for more selective communication between slabs.

The pRings are primitive vector processors which operate on data vectors delivered to them by a global communication infrastructure. Thus, the system can be viewed as a parallel processor whose components are parallel vector processors. The task of the global communication system is to transport data vectors to the vector processors. The complexity of the global communication system is then a function of the number of pRings (M) in the system rather than the number of PEs (K) in the system. The reduction in complexity is then some function of the number of PEs in a pRing (k) which can be several hundred. It is therefore possible to consider traditionally poor scaling topologies in terms of cost such as cross-bar and full interconnect, although modularity and extensibility may be sacrificed.

If the number of logical slabs in the ANN model is small and the number of pRings in the MNR implementation is large, then each slab can be assigned to several pRings. In this case, the pRings can be connected in a string and contiguous pRings are assigned to a logical slab. This connection and assignment scenario is a recurring theme in ANNs and suggests string connections between pRings.

In effect, several pRings strung together as such are viewed as a larger, virtual neural ring if a path is provided between the end pRings in the string. However the location of this feedback path is dependent on the particular ANN model being implemented. If all possible feedback paths were provided, the connection topology can degenerate to full, point-to-point connectivity whose hardware complexity scales poorly with M (order M^2) and is not easily extendable. There are other connection topologies such as toroid, hypercube and many multistage networks, which scale much better, that can be used.

In addition to providing feedback paths to close virtual neural rings, the global communication structure preferably also provides for communication between slabs. This communication task is more demanding since there is little additional communication regularity that can be extracted among the various ANN models.

When analyzing candidate communication structures, two characteristics of the communicating pRings are important to consider. They both stem from the fact that while massive numbers of PEs are employed, they are already organized into moderately sized groups, i.e., the pRings. This effectively reduces the communication problem from that of interconnecting K processors to the smaller problem of interconnecting M=K/k processors. Whereas realistic values of K can be in the 10^3 to 10^4 range, M is more likely to be limited to the 10 to 100 range. It is no longer necessary to exclude N^2 complexity topologies from consideration.

The second characteristic is that communication between pRings involve vectors of size k. This reduces the importance of address overhead costs and improves the case for circuit switched rather than packet switched networks. It also reduces the importance of network diameter since the diameter affects the latency more than the throughput. Effects of diameter on throughput are more indirect, via congestion. Since the communication is overlapped with the processing a small increase in communication bandwidth will overcome the effects of latency. That is, once a trail is blazed from a source to a destination pRing, the data vector flows at the full burst communication rate.

Although many communication structures can be used to augment the pRing string structure, a bus is preferably used to exemplify an MNR architecture as depicted in Fig. 4. The bus, together with the string communication structure provides the communication infrastructure for the Bussed Modular Neural Ring Architecture (BMNR). The bus hardware requirements scale linearly with M and pRings can be modularly added to increase the system size without modifying each pRing (the communication degree of the each pRing is constant). The communication time is approximately constant except for the small overhead for communication set up.

The MNR architectures contain a number of pRings, each of which is controlled by its own, potentially unique, program. At the pRing level, processing proceeds in a synchronous, SIMD fashion. At the system level, however, the pRings are more loosely coupled executing separate, asynchronous programs. Overall system synchronization is attained partially by the pRing programmer and partially by asynchronous handshaking between pRings. Thus, processing at the subvector level proceeds in systolic array

fashion while system level processing is more asynchronous in nature.

As discussed earlier, appropriate choices of SIMD array size (k) versus number of vector processors (M), for ANN implementations with sufficient local processing and communication regularity, can lead to flexible as well as efficient systems. Fully SIMD systems suffer utilization loss due to fragmentation while fully MIMD systems incur large control cost overheads.

The core of the MNR architecture is the pRing. A pRing block diagram is shown in Fig. 5. As discussed above, a pRing consists of a number of primitive processing elements (PEs) 54,56, 58 and 62 served by a local ring communication structure 64 and managed by a local centralized control unit (CU) 66. The CU is coupled to a control memory 67. Each pRing is interfaced with the global communication system 68 via the interface unit (I/F) 70. The interface unit is also supervised by the control unit 66. For the bussed MNR architecture discussed above, the interface unit provides connections to the left and right adjacent pRings and to the bus.

The ring communication structure within a pRing is actually comprised of three concentric, independent rings: the P ring 72 for vector processing, the R ring 74 for vector receiving and the T ring 76 for vector transmission. These rings are formed from three registers 78, 80 and 82 at the entry to each PE. The processing ring, made up of the P buffer associated with the PE's, is used to circulate an input vector to all of the PEs on the ring. This circulation occurs at the PE processing rate. The R buffers are used as an input vector receiver and staging area, so that when the PEs have finished using the vector in the P ring they can immediately begin working on the received vector from the R ring. Thus the communication of vectors between pRings can be completely overlapped with the processing. The R buffers form a chain rather than a ring, since there is no need for the received data to circulate within a receiver ring. The final set of buffers form the transmission chain rather than a ring. This chain is used to hold a data vector for transmission to another pRing.

The processing elements themselves are preferably configured to be simple. They are all controlled in parallel by a central control unit within each pRing. Thus, only the actual data manipulation parts of the arithmetic unit need to be replicated for each PE,

resulting in substantial hardware savings in implementation of a pRing. This also allows for speed versus space trade-off between parallel and serial arithmetic unit to be made without incurring the control overhead costs associated with the more serial approaches to each PE. Parallel arithmetic is the fastest and uses the most hardware, while the opposite is true for bit serial arithmetic. Since the control overhead costs have been minimized by the central controller, the time-space product remains fairly constant. Thus, the low speed of the serial PE is compensated for by the large number of PE's which could be placed on a chip. The advantage to parallel arithmetic is that the decrease in the number of PEs leads to somewhat better utilization, due to a reduction in fragmentation.

The serial PE, on the other hand, lessens the communication bandwidth problem by increasing the time spent processing an individual data vector. More importantly, with bit serial arithmetic, a time and space versus precision trade-off can be made possible and easy with programming. Thus, weight memory is bit-addressable and significant savings in memory can be achieved. Further, lower precision weights can become adequate for a given model. A similar argument holds for trading speed with precision. For large models, the fragmentation issue is of less significance and the advantages offered by serial arithmetic are substantial.

As shown in Fig. 5, pRing operation is controlled by the pRing control unit (CU) 66. The CU executes a program stored in the attached control memory (CM) 67. The control unit actually comprises three individual controllers and a control memory 84 as shown in Fig. 6.

The master control unit (MCU) is responsible for fetching instructions from the control memory and either executing them directly or dispatching them to the other control units. As will be described below, the MCU also contains a micro-processor and a number of registers, which are used for a scratchpad and housekeeping for executing a pRing program. Inter-pRing communication instructions are forwarded to the interface control unit (ICU) which has the responsibility of controlling data vector transmission and reception. PE instructions are dispatched to the PE control unit (PCU) which decodes them and broadcasts control sequences to all of the attached PEs. The PE control unit can execute "step and repeat" instructions which allow it to autonomously process an entire

vector circulation cycle. Instruction execution in each control unit is overlapped with execution in the other two. The MCU, however, performs instruction fetching for all three controllers. This configuration is not a bottleneck, because the ICU and PCU instructions typically operate on entire vectors of data.

Fig. 7 is a block diagram of one embodiment of the MCU (and attached control memory. It is organized much like a general-purpose microprocessor with the PCU and ICU operating essentially as attached co-processors. The registers 94 and ALU 96 shown preferably do not participate directly in the vector data processing performed by the pRing. Instead, they are used to control operation of the pRing program. Local instructions are latched into the instruction register for the MCU and carried out by the MCU. Instructions destined for the ICU or PCU are automatically sent to the appropriate controller. Also, operands required for these instructions are automatically fetched by the MCU and forwarded to the destination controller. Synchronization instructions are provided to ensure that the ICU and PCU are not overrun with instructions.

As discussed above, the pRing is controlled by a program which resides in the control memory, is fetched by the MCU and is executed on the MCU, PCU and ICU. An instruction set, an instruction encoding format and an assembly language for the pRing is provided in accordance with the present invention. The instruction set and machine code definitions are provided in Appendix A. The programmer's model of the pRing is given in Figure 8. In this representation, the programmer has direct access to the local MCU registers 94, ALU 96 and control memory 84. The PEs are viewed as a processing subsystem over which the programmer has control but no direct access to the data. Similarly, the programmer can control the interface unit but cannot directly access it.

The pRing instruction set looks largely like that of a typical microprocessor. But in addition to the usual instructions, special instructions for controlling the PCU and ICU are also included. These instructions include step-and-repeat forms for the PCU and vector-block forms for the ICU. Conditional transfer instructions can be used to test the status of the other two control units and thus provide a level of programmed synchronization between the units. The job of the pRing program is to choreograph vector processing and system communication at the pRing level.

The instruction set is divided into three groups: local instructions, PE instructions and interface instructions. This division is reflected by a two bit field in the opcode field of the instruction format. The MCU monitors this field for passing each instruction to the appropriate controller.

There are five addressing modes, which are made available for most instructions. These are immediate, direct, register, register indirect and memory indirect. When fetching operands for ICU and PCU instructions, these operands are never used directly as data. Instead they are used as instruction parameters such as weight memory offsets and base addresses.

Due to the diverse nature of the available ANN implementation architectures, comparing the performance of one to another is very difficult. However, DARPA used two generalized variables, referring to speed and capacity, as assessment criteria on the graph depicted in Fig. 9. Capacity is measured by neuron-interconnections, indicating the size of the ANN model which is implemented. Speed is measured by interconnections-per-second, indicating the rate of primitive ANN operations (i.e., multiply-accumulate). On the speed/capacity plane, fully parallel architectures 104 reside in the upper left corner. Serial CPUs 106 are positioned in the lower half of the plane (Figure 9). Fully parallel architectures may perform at very high speed but are severely limited in capacity. On the other hand, serial CPUs are limited in speed but they are quite expandable in capacity (provided that there is adequate memory available). The MNR architecture 108 is designed to compromise performance and to fill the gap between fully parallel implementation architectures and simulation on serial CPUs. Simulation results described below substantiate that the performance of the MNR architecture resides in the diagonal region between the fully parallel implementation architectures and the simulation on commercial digital computers.

MNR ARCHITECTURE PROTOTYPE

One of many possible MNR architecture prototypes, which has been designed and constructed using off the shelf discrete integrated circuit components, will now be described. A production modular pRing neural computer can also be designed and

implemented using custom or semi-custom VLSI components to reduce size and power requirements and to increase performance. The production unit of a modular pRing neural computer is somewhat different from what is described herein when VLSI components are used. The required design deviations, however, do not affect deleteriously the MNR architecture's capabilities.

A. Overall System Level Design

The purpose of implementing the prototype described herein is three fold. First, the prototype serves as a proof of concept model by demonstrating, from a real engineering perspective, that the architecture is practical. The prototype also provides a vehicle for further study of the problems associated with such architectures so that a commercial version can reap the benefits of engineering decisions made on the basis of data from a real machine. Finally, the prototype allows true performance measurements to be taken with an accuracy that can not be achieved through simulation and analysis.

The goal is not to produce the fastest, the most efficient, the most flexible or even the most elegant design. The prototype is simply an evaluation and test platform for architectures in the MNR family.

As such the prototype is of modest size and is implemented with mature (if not, in some cases, old) technology. The prototype consists of five pRings with up to 40 PE each for a total of 200 PEs which is much smaller than desired for a high performance commercial product. The PEs perform low level operations at the rate of 200 ns per bit which is far from state of the art. Much of the design uses fairly low complexity programmable logic arrays (PLAs). Thus the performance measurements taken from the prototype are somewhat low. These numbers need to be scaled to bring them in line with state of the art commercial technology and more respectable system size.

An ANN implementation workstation was implemented. In this system, the MNR subsystem is connected as a coprocessor 112 to a general purpose host computer 114 as shown in Fig.10. The host computer 114, i.e., an 80386 class personal computer (PC), serves as the user interface for the ANN workstation and hosts the ANN software development tools. It is also responsible for downloading (to the MNR coprocessor) initial weight values, pRing programs and data as well as supervising the overall operation

of the MNR system.

Within the MNR subsystem is a number of pRings 116, 118 and 120, each of which is connected to its two immediate neighbors and connected to the MNR bus 122. The adjacent pRing connections allow several pRings to be assigned to a larger virtual ring and they support adjacent layer communication in certain layered models. This is shown in Fig. 11. The bus connection is provided for more general intra-pRing communication requirements and for closing the loop in larger neural rings constructed from multiple adjacent pRings. The communication bandwidth of the bus, therefore, is a function of the degree of local specialization and intra-slab communication and not a function of the absolute ANN size. In fact, as it happens, the larger the ANN model, for fixed MNR system and fixed ANN architecture, the lower the communication bandwidth requirements.

pRings are interconnected via ribbon cables, and the Host to MNR coprocessor interface is preferably provided by a special purpose interface cards (not shown). The interface card allows the host computer to control the coprocessor operation by allowing the host to become an MNR bus member.

The core of the prototype system, and for that matter, any MNR architecture, is the pRing. Fig. 12 is a block diagram of a pRing 116 for the BMNR workstation. It consists of three centralized control units and a number of primitive processing elements (PEs) connected in a string. The string of PEs 128, 130, 132 and 136 emanates from and terminates in the Interface Control Unit (ICU) 138 where the string can be closed to form a ring 140.

Each PE is a relatively primitive processor (i.e., only an ALU) with associated weight and accumulator memory as well as a stage of the communication string. These stages form a collection of shift registers with the shift dimension being along the PE string. The shift registers so formed are used for data vector reception, transmission and circulation for processing. Reception, transmission and circulation operations can occur simultaneously and independently so that communication and processing can be overlapped. Note that the shift register file is situated outside the ALU so that data need not pass through the processing circuits as is often done with systolic arrays.

The centralized controllers in the pRing are the Master Control Unit (MCU) 142, the Processor Control Unit (PCU) 144 and the Interface Control Unit (ICU) 138. The MCU controls the overall operation of the pRing and it is this unit that executes the pRing program. The MCU controls the ICU and directs the instructions to the PCU.

The PCU (Processor Control Unit) 144 has direct control over the processing elements and their attached weight and accumulator memories. It receives a stream of macro-instructions from the MCU and decodes these instructions into control signal and memory address sequences which it broadcasts, in parallel, to all PEs in the pRing. It also directly controls the shift register file and provides arbitration and handshaking for the ICU to access the shift registers for communication purposes.

The ICU (Interface Control Unit) is responsible for vector communication with other pRings. It controls data vector transmission and reception in the PE shift register file indirectly through the PCU. This is the unit most affected by the global communication structure.

Photo 1 shows a prototype pRing used in the system. The PCU serves as the mother board for the pRing, hosting up to five PE string boards oriented vertically on one end of the PCU. The MCU and ICU can be combined into one PCB (printed circuit board), the MICU (Master and Interface Control Unit), which is "piggy-backed" on top of, and parallel to, the PCU.

B. The PE String Module

The PE string board contains eight processing elements (PEs) including their associated accumulator and weight memories and shift register circuitry. Fig. 13 is a block diagram of the PE string board. PE string boards are designed to be concatenated to form arbitrarily long strings of PEs. As shown in Fig. 13, parallel address and control lines are used for all PEs and one bit is used to cascade PE string boards to form larger chains.

The core of the PE string board is the array of eight Processor Logic Blocks (PLB) 150 that perform the computation. Each PLB is an ALU for one PE. Fig.14 shows the detail of a PLB. Logically, the PLB consists of a small number of one bit registers, a data exchanger 152 and an ALU 154. The registers include flag registers C (carry) 156 and Z (zero) 158 as well as three operand registers Y, Q and WD indicated by 160, 162 and 164

respectively. The data exchanger allows the data movement operations listed in Table 1. The ALU implements the bit operations listed in Table 2.

Table 2. Operation Code Table

C OOOO DPPP E210	LOAD PHASE OP3*PHASE Y	WO	OPERATE PHASE OP3*/PHASE AD	C	INIT /OP3 /PHASE Q	C
0000	A	Q*WI	Y*WO	0	S	0
1001	A	WI	Y*WO	0	C	0
2010	A	Y	Y*WO^C	Y*WO+Y*C+WO*C	Z	0
3011	A	Q	Y*WO^C	/Y*WO+/Y*C+WO*C	Y	0
4100	/Q*A+Q*WI	1	Y*WO	0	WI	0
5101	/Q*A	/Q	Y*WO	0	Q	0
6110	Q*A+/Q*WI	1	Y*WO^C	0	0	1
7111	Q*A	Q	Y*WO^C	C	Y	C

Table 2. Exchange Code Table

C OOOO DPPP E210	/OP3 AD	Y	WDO	XD /OP3	OP3
0000	Y	AD	AD	SD	SD
1001	Y	AD	SD	WDI	WDI
2010	SD	AD	Y	AD	AD
3011	SD	AD	Q	WDI	WDI
4100	Y	WI	AD	Q	Q
5101	Y	SD	SD	WDO	WDO
6110	SD	WI	Y	Y	Y
7111	SD	SD	Q	C	C

The PLB is actually implemented in a field programmable logic array (PLA) so its physical structure is not as depicted in the figure. Also, certain limitations were imposed by

the limited number of product terms (7 or 8) available in each output logic macrocell. However, the operations listed in the table are sufficient to implement the instruction set of a pRing.

The pRing block diagram shown in Fig. 12 calls for at least three shift register stages per PE where a shift register stage can hold one, maximum precision, data element. The maximum precision originally chosen was 16 bits. This would require 48 bits of shift register for each PE. Implementation using discrete components would have required six chips (8 bit shift registers) per PE for a total of 48 shift register chips on the PE string board. This is excessive when compared to the only 11 chips used for everything else on the board.

Thus a shift register simulation scheme shown in Fig. 15 is used in accordance with the invention which implements all of the shift registers using only four chips including a memory 166 and a latch 168. The large chip count for the direct shift register implementation is due to the off-the-shelf, discrete component constraint. In a VLSI implementation, this would not be as serious a problem. However, the RAM based shift register technique bestows other advantages on the design. First, the precision need no longer be limited by the length of the shift registers since RAM is so economically fabricated (compared to direct implementation shift registers). Next, there is no longer a requirement for hard partitioning of the shift registers in to the three types required. Instead, the shift register memory forms a shift register file which can be partitioned in software into the number and type of virtual shift registers needed. Finally, when implementing lower precision operations (such as multiply) there is no need to waste clock cycles by clocking an operand to the head of a register. With this implementation, the shift register file consists of 256 one bit shift registers that can be partitioned into various sizes and numbers of word shift registers. Operand widths of up to 64 bits are easily accommodated.

The main disadvantage of this scheme is that more complex control circuitry is required to implement the shift register simulation than to implement shift registers directly. But this control circuitry is manifested as a one time cost in the PCU, rather than a recurring cost for each PE, so even with only one PE string board, there is still a net savings. A small penalty in speed is paid for this expedient, but this is not the critical path in the system so it is affordable.

Each PE contains a one bit ALU with the instruction set repertoire is given in Tables 1 and 2. Word level operations are accomplished by multiple bit level operations as in any bit serial ALU. The control for these concatenated control sequences is provided by the PCU, thus the serial ALU control overheads, which are a one time cost in the PCU, are amortized all PEs in a pRing. Rather than using shift registers for operands and results as is usually done in serial ALUs, the data memories 166 are, themselves, used as shift registers by cycling the addresses appropriately. If a single operand memory were used instead of separate weight and accumulator memories, two reads and a write would be required for each bit operation, thus the memory is broken into a large weight memory and a small accumulator memory. But this still creates a speed bottle-neck at the accumulator memory which must be accessed for one read and one write in most operations. For this reason, a small, fast memory is chosen for the accumulator which can be cycled twice in the time required to cycle the weight memory. The weight memory is necessarily the largest memory since the number of weights dominates the size of the ANN.

The memories are bit addressable, so that maximum use can be made of available memory resources. Models requiring less precision, then, benefit in both time and number of connections available. The maximum precision in this implementation is 64 bits.

C. The Processor Control Unit (PCU)

The Processor Control Unit (PCU) controls the processing elements (PE) in a pRing. It is a single board with connectors on one end for the PE string daughter boards and an interface to which MICU (Master and Interface Control Unit). The PCU can host up to five PE string daughter boards for a total of 40 PEs. The function of the PCU is to emit addresses and control signals to the attached PE strings in order to effect the processor instruction set. The PCU requests macro instructions and parameters from the MCU and carries these out by sending address and control sequences to the PEs. It also has direct control over the PE string boards' shift register logic for transmission, reception and circulation of data vectors. The ICU, however, is actually responsible for sending and receiving data vectors from other pRings.

Fig. 16 shows the major logical blocks of the PCU. They include the clock and strobe generation block 172, instruction memory and sequencer 174, the PE interface 176, the

MCU/ICU interface 178 and the parameter memory and accumulator 180. Although there is some overlap in these units in the actual implementation, logically this is a good decomposition.

The core of the PCU is the instruction memory and sequencer 174. The PCU is configured as a microcoded controller with a writable control store (WCS) 182. There is preferably no read only instruction memory so the controller must be boot strap loaded by the MCU. All PCU instructions are single, 40 bit words. Table 3 summarizes the instruction word format.

Table 3. PCU Instruction Word Format

BIT	GROUP0		GROUP1
0		IMEDOE	
1		AASEL	
2	ACS		PCS
3	PHASE		PRMOP0
4	PCLK		PRMOP1
5	AWE		PWE
6	INCWA		LDWA
7		AOE	
8..1	OP(0..3)		PAD(0..3)
12..14		SHOP(0..2)	
15	GROUP0		GROUP1
16	INCAA		LDAA
17	WOP0		PINSEL0
18	WOP1		PINSEL1
19		NOT USED	
20		NOT USED	
21..23		PCOP(0..2)	
24..39		IMED(0..15)	

	WOP(0..1)		SHOP(0..2)
00	-	100	Read Shift Memory
01	Read Weight Memory	101	Write Shift memory
10	Write Weight Memory	110	Read&latch shift data
11	-	111	Latch XD (exchg data)
	PCOP(0..2)		PCOP(0..2)
000	INC	100	JPCY
001	JMP	101	-
010	DJZ	110	DECCTR
011	JPACK	111	LDCTR

	PINSEL		PRMOP(0..1)
00	Counter Select (CTR2P)	00	HOLD
01	SUM Select (SUM2P)	01	LOAD ADDER VALUE
10	External Select (EXT2P)	10	COUNT
11	SUM2P&OUT PRQ (IMD15)	11	LOAD IMMEDIATE VALUE

Each word includes a 16 bit immediate data field and 24 additional bits for various control functions. The control store accommodates 2048 forty bit instructions. It is sequenced by an 11 bit program counter (PC) and it has a 2048 word instruction memory. The PC is augmented by a loop counter and condition code selector. In addition to the default increment of the PC, it can also be loaded from the immediate bus to implement conditional and unconditional jumps. The jump conditions are PCU status information such as the loop counter being zero, and they do not generally include status about data being processed in the PEs. The immediate bus can be driven by either the instruction memory or by the parameter memory under instruction control. Thus indirect jumping and subroutine linkage using parameter memory locations is available. This also allows for dispatching microcode routines from instruction addresses specified by the MCU. These addresses define the macro instructions as seen by the MCU. It is important to note that the application programmer need never program this complex unit. Instead, the PCU is used as a controller to give the pRing its data processing instruction set.

A loop counter is provided for iterating instruction loops. The counter size was chosen as size bits. This is the source of the 64 bit precision limitation of the pRing. Since the PEs are implemented as serial ALUs using RAM instead of actual, fixed size shift registers, they do not limit the precisions. Greater than 64 bit precision could actually be implemented by double loops in the PCU microcode.

The PE interface contains the address counters and control signal conditioning logic to drive the PEs. An address counter is provided for the accumulator memory address bus and another is provided for the weight memory address bus. The weight address counter can be gated onto the accumulator address bus for instructions involving two accumulator addresses. The on board accumulator latch/counter bus is used for the PCU to supply shift register memory addresses. The PCU's on board accumulator is distinct from the accumulators associated with the PEs.

The MCU/ICU interface block 178 includes logic for downloading the PCU microcode, logic for receiving macro instructions and parameters, and logic for allowing the ICU to access the PE send and receive shift registers. Access to the WCS 182by the MCU for microcode download is provided by a programmed I/O path from the MCU. A 16 bit parameter latch is provided so that the MCU can deliver parameters and microinstruction subroutine addresses to the PCU via parameter memory. A fully interlocked handshake mechanism is provided for this interface using a parameter request (PRQ) signal emitted, under program control, by the PCU, and parameter acknowledge returned by the MCU. This mechanism allows for a microcoded instruction fetch routine to read an instruction into the local parameter memory where it can be executed.

Access to the shift register memory on the PEs by the ICU is provided by a shift arbiter and multiplexing logic. The PCU is given priority for shift register access and an asynchronous handshake mechanism is implemented for the ICU interface. The PCU has preemptive priority to the shift registers but because of the relatively infrequent access by the PCU, this does not have much impact on communication of data vectors between pRings.

The parameter and accumulator block 180 comprises a 16 word by 16 bit read/write memory, a sixteen bit adder and a counter/latch for an accumulator. This block is used for PCU scratch pad as well as for receiving and storing instructions and parameters from the MCU. The main purpose for the accumulator is to add user specified offsets to weight addresses when executing various supervector instructions.

The PCU instruction word is very horizontally encoded. As such, many control strobes can be simultaneously asserted and several diverse microoperations can be accomplished in the same instruction cycle. The assembly language allows the expression of such instruction parallelism using an overlay syntax. An assembler that allows instruction overlays and, to a small extent, performs instruction bit conflict detection has been implemented as part of this work. However, the microcode for the PCU is complex and generally requires an intimate knowledge of the PCU hardware to write. Fortunately, however, the pRing programmer will not program the PCU at this level. Instead, microcode subroutines for higher level macro instructions will be invoked. The PCU system programmer needs to program the PCU in order to implement the PE macro instruction set

as seen by the pRing programmer.

D. The Master Control Unit (MCU)

The MICU (Master/Interface Control Unit) 184 (Fig. 16)is the combination of the MCU (Master Control Unit) and the ICU (Interface Control Unit) on a single PCB. This board 184 is piggy-backed on top of the PCU (Processor Control Unit). Photo 1 shows how the MICU fits in the system. The MCU and ICU were combined onto the same board for physical packaging reasons, however, they are logically distinct units. As such the MICU will be described by describing the MCU and ICU modules separately.

The MCU (Master Control Unit) is the central controller for the pRing and is depicted in Rig. 17. It is preferably a standard 80186 microprocessor 180 design. The MCU has 64K bytes of EPROM 190, 64K bytes of RAM 192, two RS232 asynchronous serial interfaces 194 and 196 and interfaces to the ICU and PCU 198 and 200. The EPROM contains MCU bootstrap initialization code as well as an assembly level debugger (DBG86) and a remote debugger kernel (Paradigm's TDREMOTE) for use with Borland's source level, remote debugger. Until the availability of ANN level languages and compilers for the pRing architectures, the MCU is the level at which the pRing is programmed. In order to minimize the programming burden, the MCU is preferably implemented with the ubiquitous 8086 family processor and has been configured for programming in the C language. The resident debugger and the remote debugger support tremendously reduce the software development task.

A serial port 202 is connected to the development computer 204 and is used for downloading and executing code developed there. This can be done by using the resident DBG86 and a terminal program or by using the remote symbolic debugger. A switch selects which resident program executes after a reset. The other serial port was provided for individual pRing status displays, but, with the advent of a system level debug tool, it is now seldom used.

The MCU is responsible for providing an instruction stream of PCU macro instructions during system operation. It also bootstrap loads the PCU microcode control program into the PCU's WCS (Writable Control Store). The instruction stream interface is a DMA (Direct Memory Access) channel between the PCU and the MCU. The asynchronous

PCU instruction and parameter interface is described in the earlier section on the PCU. Although the interface supports programmed I/O, the DMA interface is much faster and allows the MCU to attend to other overhead chores while instruction blocks are fetched by the PCU directly from the MCU's memory.

The MCU to ICU interface is handled using a small set of I/O locations and two interrupts. Although the interface is implemented using programmed I/O, the ICU is sufficiently autonomous to limit MCU interaction to pRing sized subvectors.

E. The Interface Control Unit (ICU)

The other major subsystem on the MICU is the ICU (Interface Control Unit). This unit, depicted in Fig. 18, is responsible for pRing to pRing communication. It is initialized and supervised by the MCU through the interface described above. The ICU interfaces to the PCU to obtain data for send operations and to deposit data during receive operations.

The ICU to PCU interface protocol is preferably an asynchronous, fully interlocked handshake. The PCU does not distinguish send from receive operations, i.e., it always shifts the PE data by one bit from least to most significant bit. Recall that the shift registers reside on the PE string boards but are controlled by the PCU. Thus, the ICU interfaces to the PCU where arbitration and control for shift register access is implemented.

Send and receive operations are named from the point of view of the ICU. Thus, if the ICU is performing a receive operation, it supplies data to the PCU and ignores the value returned by the PCU. For send operations, the ICU uses the bit supplied by the PCU and does not provides data on the PCU input bit. PE shift register addresses are provided by the ICU.

The primitive operation performed by the PCU for the ICU, then, is just shift one bit (along the PE string dimension) of the 256 bit shift registers available. Thus the logical organization of the shift register file bits into words is accomplished by the ICU. However, the ICU itself imposes some constraints on the use of the shift register file. The ICU organizes the shift register file into sixteen registers of up to sixteen bits each with lower precision words justified to the low bits of the registers. This allows for some simplification of the ICU hardware without an appreciable loss in flexibility. When the data vector precision exceeds sixteen bits (up to 64 is supported), the data is sent in 16 bit chunks.

There are four send destinations and the same four receive sources. Two of these are the adjacent pRings in the string. Another source/destination is the MNR bus. The final port is the MCU. The MCU data path is provided for bootstrap and debug purposes. The ICU can send to, or receive from, a single device at a time. However, send and receive operation can occur simultaneously with independent source and destination with different shift register file addresses. Arbitration for use of the PCU interface between send and receive operations is performed by the ICU on a first come first served basis with ties being granted to the last operation that did not use the interface. In this manner, lockouts are avoided.

F. The MNR Bus

With reference to Fig. 19 the MNR bus is preferably a multiple access, multi-master, distributed arbitration, handshaked bus. It uses one clock line (BCLK) and one, open collector data line (BUSIO). All pRings are synchronized to the common bus clock which is provided from a single master source. Each pRing has its own local oscillator which provides the pRing's ICU clock. Thus some synchronization between the bus subsystem and the remainder of the ICU in each pRing is required. This is accomplished using a variety of techniques (D2,D4).

The MNR bus is unique in its use of protocols for reducing the required number of signals. Using only a clock and one signal line (named BUSIO), the bus implements a multi-master, receiver addressed handshaked data transfer protocol with distributed arbitration and lockout avoidance protocol. This is accomplished by using the one signal wire differently in each phase of bus usage. Using this mechanism, no signal bandwidth is lost as would be the case with several special purpose signals. For example, since communication is performed with vectors between pRings, the bandwidth of separate address lines would be wasted during the vector transfer after the receiver was identified. Similarly, bandwidth would be wasted on separate signals for arbitration or data handshaking. The bandwidth of these wires could be much more efficiently utilized by implementing multiple busses with the marginal cost of only one wire per added bus (using the same global bus clock).

Transmissions on the MNR bus are from a single source (called the master) to a single destination (called the receiver). A bus transmission consists of five phases which are IDLE, START, ARBITRATION, RECEIVER ID and DATA as shown in Fig. 19. Fig. 20 is a

simplified state diagram which shows the protocol for using the bus.

The first phase 226, IDLE, is not really a transmission phase since it is simply the idle state of the bus when no transmission is taking place. In this phase, BUSIO 218 (Fig. 19) is high (that is, it is not pulled down by the open collector devices connected to it). At any time during the IDLE phase, any pRing can pull BUSIO low to signal its intention to use the bus. This is called the START 228 phase and lasts for exactly one bus clock period. Since the bus is a single, open collector line, several pRings can, in fact, request use of the bus simultaneously in this manner. This is sorted out in the ensuing ARBITRATION phase 220.

In the ARBITRATION phase, arbitration between potential senders (masters) is performed. Only pRings that are ready to send at the start of this phase may compete for the bus. Each pRing has a four bit bus address. Arbitration is accomplished by performing a distributed binary exchange search for the lowest addressed competing master. Thus pRings with lower addresses have higher priority for mastership of the bus.

Contention resolution is performed in four bus clock cycles as follows. All potential senders that are ready at the start of the ARBITRATION phase 230 will participate in the first arbitration clock cycle by placing the most significant bit of their board address on the bus by pulling BUSIO low if their address bit is 0, otherwise not driving the bus. At the end of this clock cycle, pRings whose high address bit are different from the value on BUSIO drop out of the competition. Because of the open collector wire ANDing on the bus, pRings with a 0 in that address line remain on the bus. A pRing with a 1 in the high address bit remains on the bus if no pRings with a 0 in that bit were competing. The surviving contenders from the first round then perform the same sequence with the next most significant address bit. This process continues, eliminating lower priority (higher addressed) senders at each stage until, after the fourth clock, only one sender remains which is then the bus master.

The next phase, RECEIVER ID 232, is used for the sender to identify the receiver by the receiver's board address. This address is placed on the bus in four consecutive clocks starting with the most significant address bit.

After the receiver has been identified, the DATA phase of the transmission occurs. This actually consists of the three sub phases RXREADY, DATA and TXREADY. In the clock cycle immediately following RECEIVER ID, the receiver will indicate its readiness to

accept data by pulling BUSIO low. On this first RXREADY subphase 234, the receiver should be ready or the transmission is aborted (with no data sent). If the receiver is ready, the sender will place a bit of data on the bus in the next subphase (DATA). After this, the sender indicates its readiness to transmit another bit by pulling BUSIO low (this is the TXREADY phase 236). The sender has up to two bit times to become ready before the transmission is aborted. After the TXREADY phase 236 (assuming the sender indicated readiness to send) the RXREADY phase begins now giving the receiver up to two clocks (instead of one clock as on the first RXREADY phase) to indicate readiness to receive before transmission abortion. Thus, a bit of data can take 3, 4 or 5 clocks to communicate in the steady state. This process continues until either sender or receiver causes a transmission halt by not indicating readiness in its respective phase. Usually, this will happen at the end of a send block when the sending unit has no more data to transmit. However, it is possible for activity in the PCU and/or ICU of either the sender or the receiver to cause interruptions in the transmission. In these cases, the transmission simply starts again from where it was interrupted. The sender's and the receiver's counters maintain the current position status for the block.

After the DATA phase, the bus normally the returns to the IDLE state 226 where BUSIO is pulled high. If, however, another sender has been waiting to use the bus, a separate idle cycle is not actually used. In this case, the START phase occurs immediately after the busy indication (BUSIO high) that caused the termination of the last transmission.

The bus protocol also has a mechanism that prevents the high priority devices from locking out the lower priority ones. This might otherwise be a problem if a high priority device is trying to make contact with an uninitialized receiver. The lockout avoidance protocol is simply that any master having been granted the bus cannot compete for it again until the bus has actually been idle for one clock. From the above description, an idle cycle can only occur if no sender is waiting to use the bus.

To better understand how this lockout avoidance protocol works, consider the case of senders 0, 2 and 4 all trying to send to busy receivers. On the first transmission, they all compete for use of the bus, and pRing 0 wins the arbitration since it has the lowest bus address. Upon finding a busy receiver, the transmission is terminated. Senders 2 and 4 are

both waiting to use the bus. Although sender 0 also needs to use the bus, it cannot compete in the next cycle since the bus did not go to an idle cycle after the last transmission (senders 2 and 4 kept it from going idle). Now 2 and 4 compete and this time 2 is granted the bus. Again, upon finding a busy receiver, the transmission is terminated. Now sender 4 is waiting to use the bus and neither senders 0 or 2 can compete for the bus because there have been no idle cycles since either was granted the bus. Finally, 4 is granted the bus (there was no competition) and the transmission aborts for lack of a ready receiver. Now, because all three senders have been granted the bus with no intervening idle cycles, none of them may compete for the bus. This causes a bus idle cycle which allows all three senders to, once again, compete for the bus.

Each pRing operates on its own internal clock and is, therefore, asynchronous to every other pRing in the system (at the clock level). Thus, intra-pRing communication is handshaked. Higher level synchronization is the responsibility of the pRing programmer by advanced and careful scheduling of computation and communication operations. Variations in pRing clock speeds usually result in implicit resynchronization at the subvector processing level by blocking operations in the ICU. However, system start and stop operations as well as synchronization in cases where the implicit ICU synchronization is inappropriate require another global synchronization method. This is accomplished by two open collector party line signals bussed to all pRings. Each pRing can pull the lines low and can read their state. While the use of these signals is application dependent, the following example demonstrates their use.

The two party line sync signals are independent and in this example only one is required. First the signal is brought low by all pRings (this is the default condition achieved after a reset). Then, as each pRing prepares to execute a cycle of computation (application defined), it releases its hold on the sync line and waits for the line to go high. The line will go high only when the last pRing has released it. After sufficient time for all pRings to detect the high state of the line, each pRing brings the line low again and begins its cycle of computation. When the pRing completes its cycle, it releases the sync line waits for it to go high, indicating that all pRings have completed their computation cycle.

DEVELOPMENT OF SUPPORTING SOFTWARE TOOL:

A. The Instruction Set and Programming Hierarchy

There is a hierarchy of levels at which the MNR architecture is programmed as shown in Fig. 21. At the top level, the ANN is decomposed and mapped 238 onto the pRing resources available. This process is currently manual and involves determining an efficient map from the virtual ANN processing and communication requirements to the pRings and communication scheduling. After a suitable decomposition has been determined, each pRing requires its own program. The pRing programs may be unique or several pRings may execute the same program depending on the degree of regularity in the ANN model. The pRing programs execute on the MCU and consist of a skeletal framework in which communication and vector data processing is scheduled. The pRing program running on the MCU issues communication commands 240 to the ICU and data processing macro-instructions 242 to the PCU. It ensures synchronization between processing and communication by program constructs in the pRing program. Communication commands are carried out in ICU 244 by hard wired state machines. The data processing macro-instruction stream sent to the PCU 246 is decoded and interpreted by a PCU micro-code program. This micro-code program causes the PCU to broadcast addresses and control signals to all attached PEs in a pRing. The addresses are used to fetch operands and store results in weight, accumulator and shift register memory. The control signals are interpreted by the PLBs on the PE string boards and used to direct the operands through the ALU.

Fig. 22 depicts the chain of the hierarchy that deals with data processing. Overall control 248 is provided by the pRing program executing on the MCU. This program is written is C. Blocks of macro-instructions to carry out various phases of processing are set up in the MCU's memory. These blocks are created either in advance or during ANN execution by a set of C macros which, when used in the program, have the appearance of an assembly language program embedded in the C code. The macro-instruction blocks 250 are sent to the PCU via a DMA channel. More accurately, the PCU fetches these instructions via the DMA channel. When an instruction and its parameters have been fetched by the PCU, the PCU executes a micro-code subroutine that implements it. During the execution of the micro-code subroutine, the PCU can emit addresses 252 and control signals to the

attached PEs.

The PEs interpret the control signals as opcodes 254 for single bit operations which are carried out on addressed operands by the PLBs. Every PE on a pRing executes the opcode on data at identical addresses within each PE. There are a few conditional instructions that can use local data within a PE to modify the execution of some operations. This gives rudimentary data dependent and, using tag constants, PE position dependent capabilities. These capabilities were minimized, however, in favor of a more compact and efficient PE design.

B. The PCU Microcode Assembler

The PCU is a custom designed microcoded control unit. From the requirements of PCU programs, a mnemonic PCU micro instruction set is described in accordance with the invention. Because the instruction word is horizontal, many instructions can be executed simultaneously. This limits the applicability of commercially available table driven cross-assemblers. Thus, a PCU specific assembly language and overlaying assembler (called CMDASM) are presented for development of the code. (See Appendix D)

The PCU instruction word is very horizontally encoded. As such, many control strobes can be simultaneously asserted and several diverse microoperations can be accomplished in the same instruction cycle. The assembly language allows the expression of such instruction parallelism using an overlay syntax. An assembler that allows instruction overlays and, to a small extent, performs instruction bit conflict detection has been implemented. However, the microcode for the PCU is complex and generally requires an intimate knowledge of the PCU hardware to write. Fortunately, however, the pRing programmer will not program the PCU at this level. Instead, microcode subroutines for higher level macro instructions will be invoked. The PCU system programmer needs to program the PCU in order to implement the PE macro instruction set as seen by the pRing programmer. The instruction set definition and bit patterns are defined in a header file so changes can be easily made if the need arises.

C. The MCU Control Language and Supporting Software Tools

AS stated previously, the MCU (master control unit) uses an 80186 microprocessor 188. Thus, code can be developed using readily available 8086 tools on the PC. For

example, Paradigm's Locate package can be used which modifies DOS EXE files for use in embedded 8086 application. As well as PARADIGM's TDEMOTE (a remote debugging kernel for Borland's source level debugger) to debug MCU control program at source level (C language). With the support of readily available terminal emulation programs, code for the MCU can be downloaded, executed and debugged on the MCU. A small utility (called TDX) communicates with TDREMOTE, download MCU control program (DOS EXE file) and start the execution without the need of Borland's source level debugger.

The code for the MCU is being developed mainly in C using Borland's TurboC. A set of C macros is under development to define the PCU instruction set for use within the C programs (See Appendix E). These macros will allow the C programmer to view operations on the PCU as instructions executed within the C program. They will also ease the task of transporting ANN code written for the MNR simulation to code which will execute on the prototype hardware.

Until the availability of ANN level languages and compilers for the pRing architectures, the MCU is the level at which the pRing is programmed. The MCU has been configured for programming in the C language. The resident debugger and the remote debugger support tremendously reduce the software development task.

D. System Level Supporting Software

The system software from the level of the MCU (Master Control Unit) up to the host comprises loaders and debuggers on both the MCU and the host computer as well as facilities for developing ANN application programs.

An assembly language level debugger (DBG86) is used for downloading and debugging MCU code from the host Paradigm's Turbo remote debugger kernel (TDREM) can also be used to run on the MCU. This kernel allows the use of Borland's Turbo Debugger (TD) for source level symbolic debugging of C code on the MCU. A MCU program downloader, TDX, is developed for downloading MCU control program.

A set of C language macros simplify the pRing programmer's view of the PCU. In addition, a small library of hardware dependent, low level primitive subroutines are available for use by the pRing, ANN application programmer (See Appendix F).

TD and TDREM allow extensive debug facilities on a single pRing. However, an

application in this architecture actually executes on several asynchronous pRings simultaneously. MNR system level debug tool which is more geared to application level debug of multi-pRing programs can be used, as well as a master-slave protocol for system synchronization, debug and control. In this scenario, one pRing is considered the master and all others are slaves. Each pRing has a synchronization point where it resynchronizes to all other pRings and can be directed, by the master, into a debug routine where the pRing state can be examined or modified by the master using the MNR bus. The master pRing is connected to the user's console and relays user commands and data to the slave pRings and collects user requested data from the slave pRings. The master pRing could be replaced by the host if desired which would require a mechanism for the host to communicate on the MNR bus. This will have the advantage of giving all of the resources of the host (most notably the file system and extensive memory) to the master. It will also reduce the burden on the pRing that would have been master. The slave code for this interface is anticipated to be relatively small.

For the prototype system, a 80386-based PC is used as the host computer. The reasons for this choice are the low cost, good performance and open architecture of this machine. Also factored in is wide-spread familiarity with this machine among potential users, and readily available software development tools. The host computer connected to the MNR pRing coprocessor via a high speed interface. This configuration require an added circuit board in the PC and one in the MNR coprocessor. The coprocessor will include the high speed interface card and a number of pRing boards, all connected via a backplane board. The exact number of pRings will depend on space, power and cost constraints; however, the system is designed to be expandable, so that the number of pRings is not a critical parameter for the design. It will be eventually bus loading constraints that may keep the number of pRing boards below a few dozen.

The system comprises an MNR system level debug tool designed so that Host can sit on the MNR bus acting as a bus master with all of the pRings on the MNR bus as its slaves. Each slave pRing has a synchronization point where it resynchronizes with all the other pRings and with the bus master. The bus master is able to examine or modify the state of each one of the pRings. This offers the advantage that the bus master (and so the MNR

system) can access the host computer's resources (most notably the file system and extensive memory), while maintaining the access to the resources of the MNR system.

E. MNR Simulation Software

As a first step in proving the engineering practicality of the MNR architecture, a simulation was written to test and refine it. The simulation was performed at the instruction set level and was parameterized for simulated speed.

1) Simulated MNR Architecture Model

The simulation consists of a number of simulated pRings embedded in a simulated global communication structure as described in the preceding sections. Architecture and topology files are used to specify the simulated architecture parameters and global communication topology.

Each simulated pRing has an architecture like that of Fig. 12 except that the number of ICU external interfaces is variable as a simulation parameter. The logical layout of the SIME central controller shown in Fig. 6 was retained in the simulation.

Fig. 7 is a block diagram of the simulated MCU including the pRing instruction and scratch-pad memory. The control unit (CU) depicted indicates an instruction register (IR) that holds the MCU instruction. There is a corresponding IR for the PCU and ICU. The instruction memory program counter is maintained by the MCU with the CU appropriately distributing instructions to the correct controller. Operands required for non-MCU instructions are automatically fetched by the MCU and forwarded to the requesting controller. An instruction destined for a busy controller will stop the fetch process until that controller becomes ready. In this way, synchronization among control units can be easily accomplished by the pRing programmer. One minor exception is that the ICU is decomposed into send and receive subunits, each of which operate independently. So a busy ICU send in progress will not cause fetch suspension if an ICU receive instruction is fetched. Additional instructions are provided to check for busy conditions in the PCU and ICU (the MCU will never appear busy to the pRing program) to allow for more efficient overlapping of control unit operation in more complex situations.

Fig. 8 is the programmer's model of a pRing. The SIME control unit and the pRing instruction set allows the pRing to look much like a typical, general purpose CPU with an

attached vector coprocessor and a communication channel processor. The main difference is that the data processed and communicated by the attached vector coprocessor and communication channel is not typically visible to the MCU. In fact, the MCU is provided for directing the processing and communication for the PEs via the PCU and ICU. The local data memory and MCU registers are used for housekeeping duties such as loop counting and address generation for major computation cycles. Note, that the PCU and ICU, themselves, provide counters and address generators so that the MCU need only provide control at a much higher level.

The pRing instruction set is divided into three categories - MCU instructions, PCU instructions and ICU instructions. The MCU portion of the instruction set is similar to that of a standard microprocessor including the ability to manipulate data memory and internal registers. The PCU portion of the instruction set provides for vector processing in the attached PEs. All of the PCU instructions have an additional level of indirection since they supply address information that the PCU will use to control the PEs. The ICU instructions are dispatched to the ICU subunits for sending and receiving vectors.

2) An Overview of The Simulation Program

Simulation architecture and topology files are used to specify the details of the simulated MNR system. These include such things as the number of pRings (M), the number of PEs per pRing (k), the speed of each pRing control unit, the speed of a PE, the topology of the global communication network including the speed of the links, the PE arithmetic precision and the amount of weight and accumulator memory in each PE.

The MNR topology file specifies the simulated physical connectivity of the MNR system under study. The number and type of global communication port for each pRing is specified here. The communication ports may be either private, point-to-point connections or shared bus connections. In either case, the interconnections among the pRings are also specified. Consistency checks are performed during the simulation to ensure that no communication conflicts are generated by the pRing programs. The speed of communication is also parameterized to assess the impact of communication on processing throughput. With this flexible simulation of the global communication infrastructure, various members of the MNR architecture family can be simulated. Among these is the BMNR architecture.

Because simulating a massively parallel architecture on a CPU machine is inherently slow, the simulation was parameterized to set the arithmetic precision rather than accomplishing this with serial arithmetic routines. Thus, although the architecture described has dynamically programmable arithmetic precision, the simulation actually implements this with statically selectable precision and parameterized simulated arithmetic speed. The result is that model performance under various arithmetic precisions can be observed and modeled using a lumped speed model for arithmetic while the simulation can be accomplished with relative efficiency.

The simulation is performed at the machine code level. The simulated pRings are programmed in the assembly language which is compiled using a table driven cross assembler configured for the pRing. The resulting object code is executed by the simulator.

In addition to serving as a measurement vehicle for the architecture, the simulation turned out be a useful debug tool for developing pRing programs. The user interface included mechanisms for viewing weight and accumulator memory for every PE, registers and memory for each pRing and communication status for the global communication system. The simulation monitor also allowed breakpointing and single step capability at the system, pRing and PE instruction level. Even supervector instructions could be executed one element at a time. Many of these capabilities would require much additional circuitry in actual hardware implementation. Indeed, in the early stages of the hardware prototype development, simulation runs were used to help verify the prototype operation.

The simulation programs and performance evaluation of the MNR architecture are described in more detail below.

ADVANTAGES OF THIS MNR ARCHITECTURE

A. Expansion

Expansion of the hardware which implements the MNR architecture, in order to extend the scale of the ANN, can be accomplished in one of at least two ways. First, each pRing can be augmented by inserting additional PEs into the primitive ring. Logically this can be done without limits, because of the global ring communication structure; however, the control unit fan-out should provide an upper limit. Note that this upper limit is quite high and can, itself, be extended by signal amplification. Expanding the system with this method does

not cause a bus bottleneck, because even though the data packets sent over the bus are larger (increased by, say, a factor of k), the time used to process each packet is also correspondingly larger (increased by a factor of k^2).

The system may also be expanded by adding more pRings on the bus. The usual signal loading problems are relevant also here and provide an upper limit for this type of expansion as well. But bus utilization should only increase if the number of logically specialized slabs increases, since it is mainly the inter-slab communication that requires the use of the bus.

Since the MNR architecture is modularly expandable, its expansion cost increases linearly in terms of speed and capacity. Comparably, the fully parallel architecture is not modularly expandable. Cost of modularly extensible serial CPU and MNR architecture increases linearly. Extending fully parallel architectures result in exponentially increasing cost beyond a technology related, point of VLSI-implementation density, because of massive connectivity and packaging requirements.

From the speed point of view, fully parallel architectures are the architectures most favored for small-to-medium size ANN models. But the expansion cost increases exponentially with speed due to technology constraints. The serial CPUs rely heavily on technology advances to gain speed improvements. The MNR architecture allows for modular expansion and at a linearly increasing cost without relying on technology advances. Notice that any future technology advances will benefit the cost/performance figures of the MNR architecture as well.

A more important issue in the development of ANN implementation architectures is to incorporate the ability to improve performance as ANN-system needs and resources change. Fully parallel architectures are etched in silicon during fabrication and cannot be changed in the field of application. Serial CPUs can be upgraded in capacity but the system speed is mainly fixed for a given machine. Since the MNR architecture can be extended simply and modularly, e.g. by adding more PEs, and because each PE has local resources, existing MNR systems can easily be extended in both capacity and speed.

The flexibility of an ANN implementation architecture can also be measured in terms of the classes of models realizable on the architecture and in terms of the spectrum of cost versus performance alternatives. Here again, the fully parallel architecture is very inflexible.

Serial CPUs allow the most flexibility in terms of model support, since the CPU has access to all interconnections and neuron-values. However, the cost/performance ratio for serial-CPU simulations is fixed. The MNR architecture allows great cost versus performance flexibility both in design of new systems and in the upgrading of existing ones. The architecture is slightly less universal than with serial CPU in terms of models supported. This is because of the communication structure and locally SIMD nature of the pRing. However, neural networks are based on a high degree of regularity in processing and communication. The MNR architecture provides best support for models which exhibit this regularity. The regularity of neural network models is a local property. Correspondingly, the SIMD processing structure and the ring communication structure of the MNR architecture are local properties. Thus, the architecture can support a wide variety of models, which are locally regular but exhibit regional specificity.

The MNR architecture, with its SIMD processing and ring communication structure, offers a regular, modular and expandable implementation of ANNs. The regularities in neural network models also contribute to high hardware utilization in MNR implementations. The global MIMD nature and the programmability of the pRings provide great flexibility to support a wide variety of neural network models.

Since the architecture can be decomposed into VLSI implementable building blocks, it is readily realizable using our most mature technology, and it need not wait for future technological breakthroughs.

B. Variable Precision Processing Elements

Various ANN models have diverse arithmetic precision requirements. For example, the Asynchronous Binary Hopfield model requires only one bit activation values, while other models (such as ART) are described with continuous activations. Weight precision requirements are equally varied. For example, using the .15N stable memory estimate for the Hopfield associative memory model, a 200 neuron network would require less than six bit precision for weights. On the other hand, an adaptive Back Propagation network may need several times that precision for complex error surfaces and small learning rates.

These diverse precision requirements make it difficult to choose, at priori, an appropriate storage and processing hardware precision. Most general purpose ANN execution

platforms (e.g. general purpose computer simulations, digital signal processor accelerators and "neuro-computer" coprocessors) use a high precision (typically 24 to 80 bits) floating point number format. This is sufficient precision for all ANNs likely to be run on these systems, but it is tremendous overkill for many ANN models.

The solution is to provide arithmetic units that allow the programmer to dynamically reconfigure the system for the precision required. There are two obvious ways to accomplish this. First, atomic PEs could be provided, each with the capability of some minimal precision arithmetic. For models requiring higher precision, strings of contiguous PEs could be allocated as single, more powerful PEs. This "dynamic bit slicing" technique allows the application programmer to trade the number of PEs against the speed of each PE. PE expansion ratios would be limited to a factor of perhaps 4 or eight without the addition of look ahead circuitry among grouped atomic PEs. Also, word serial multiplication is favored over flash multiplication because of its amenability to this kind of modularization.

Restricting attention to binary integers, the time-space product for a multiplication scales, approximately, as the product of the factor precisions. Ignoring control overhead offsets, this remains true whether the operation is performed in parallel (a "flash" multiplier), word serial or bit serial. The more parallel the multiplier, the more chip area that is taken whereas the more serial the implementation the more time that is taken.

However, control overhead tends to dominate the hardware complexity of a bit serial multiplier. But since the pRing is SIMD, most of the control can added as a nonrecurring cost (with respect to a PE) in the PCU. Furthermore, under the assumption of large ANN models, the system can use as many PEs as can be provided, which is not usually the case for smaller or less parallel problems.

Thus, for the prototype implementation, bit serial arithmetic was chosen. The system performance (in interconnections per second) is about the same as parallel arithmetic for fixed system cost and fixed precision. However, the fixed precision criteria is artificial since each ANN will have its own precision requirement. With fixed precision, parallel arithmetic circuits, ANN models with lower precision requirements will suffer a net decrease in effective hardware utilization. Serial arithmetic allows another dimension of flexibility in the ANN implementation. Now the ANN designer can trade speed with numeric precision. For

example, a 10 to 20 fold increase in speed is obtained over using fixed, 16 bit precision on a 200 neuron Hopfield associative memory. Weight memory is also bit addressable allowing more efficient allocation of weight memory and often a net reduction in the amount of memory required.

The "dynamic bit slicing" technique trades the number of PEs with the precision of each PE while the serial processing technique trades the speed of each PE with precision. The net effect is the same in either case - lower precision yields a correspondingly higher system speed measured in interconnections per second. The latter approach gives a simpler, lower cost and more flexible design. The advantage of the former approach, is that, for higher precision models, there are less PEs and, therefore, potentially less fragmentation.

As a final, practical implementation idea for variable precision PEs, there is a family of RAM based gate arrays available from Xilix. If the PEs were implemented using these parts, the number and precision of the PEs could be programmed by reloading the gate array architecture cells before execution of an ANN.

C. Super-Vector Instructions

Each pRing is an SIMD, vector processor. The components of the vector are the PEs. As such, every instruction broadcast to the PEs within a pRing is a vector instruction. A single add instruction is multiplied by the number of PEs in the pRing (k). In addition, a number of "Super-Vector" or extended instructions exist which perform vector operations on each component of a vector. These instructions are essentially vector instructions with "step and repeat" capability added.

For example the MAC instruction (multiply-accumulate) will multiply a weight by some value and add the result to some accumulator. This is k arithmetic operations (if multiply-accumulate is considered a single operation) for the one MAC instruction. The "super-vector" form of this instruction is XMAC (extended multiply-accumulate). This instruction performs a MAC then steps the input data vector in the ring by one PE position, steps the weight address by a user specified offset value and performs the MAC again. This process continues for up to k steps so that k^2 arithmetic operations are performed. This instruction produces k inner products and is most often used to implement, in a single instruction, a matrix-vector multiply for some submatrix of the weight matrix.

Other "super-vector" instructions are MACX which is like XMAC but the accumulation value is in motion, and XPRDCT which forms the outer product of two vectors. MACX is used when the weights are stored with the sending PEs rather than with the destination PEs as in the adaptation phase of Backpropagation. XPRDCT is used extensively in the learning phase of various ANNs.

The time spent in processing a ANN is dominated by these types of operations. Inclusion of these in the instruction set serves to speed the operations by eliminating the overhead involved in instruction fetching and housekeeping by the centralized controller. It also allows for a somewhat slower (or simpler) central controller (MCU), because higher level housekeeping chores can be accomplished during the relatively long super-vector instruction times.

SIMULATION OF THE MNR ARCHITECTURE

The simulation and evaluation of the MNR architecture for implementation of ANNs will now be discussed. The performance and trade-offs of the MNR architecture have been tested. A powerful and extensible simulation tool is presented in accordance with the invention.

SimM, for "*Sim*ulation of *M*NR", is a simulation tool to represent the MNR architecture and its operation in software modules, and to provide an experimental environment for study of architectural trade-offs and for investigation of ANN model-dependent performance analysis. SimM is also used as an MNR architecture development tool, to assist in communication conflict resolution, to debug pRing programs, to analyze hardware utilization, and to experiment with various hardware configurations. SimM can also serve as an ANN model development-tool to confirm the theoretically predicted performance of proposed new ANN models.

To accomplish the objectives stated above, SimM is be capable of reconfiguring through changes in parameters. These parameters include architectural parameters, topology parameters and pRing control programs.

SimM also provides a set of on-line commands. One can operate SimM as if operating a real hardware-implemented MNR machine. This tool allows the developer to

examine and to change the values in the accumulator-memory and in the weight memory, to monitor the changing ANN states, and to check current utilization of devices. With SimM, one can test a new ANN model on the MNR implementation system as well as to test a new MNR design for an existing ANN model.

Architectural parameters are used to define the physical architecture of the proposed MNR system. The available architectural parameters determine the flexibility of SimM as a MNR architectural simulator. The activities within SimM are regulated by the architectural parameters as if they were regulated by the physical MNR architecture. The architectural parameters supported by SimM include:

1. Number of pRings on system (M)
2. Number of PEs per pRing ($k=K/M$)
3. Size of weight memory per PE
4. Size of accumulator-memory per PE
5. Number of communication ports for each pRing
6. Channel bandwidth of communication links (B)
7. Arithmetic precision of PEs (P)
8. Relative system clock cycle, in nanoseconds (ns)
9. pRing control unit (MCU) speed as a multiple of relative system clock
10. PE speed as a multiple of relative system clock
11. Interface control unit (ICU) speed as a multiple of relative system clock

The topology parameters define the global pRing communication topology of the proposed MNR system. The MNR architecture involves two kinds of communication: single-destination and multiple-destination communication. With single-destination communication, the pRing communicates via the communication port indicated by the pRing control program with the other pRing. The pRing does not know the real identity of the corresponding pRing. The identities are defined by the topology parameter, which represents the real hardware connection between two pRings. Multiple-destination communication, on the other hand, involves one sending pRing and several receiving pRings. The sending pRing is the bus master while sending data, and the receiving pRings are bus slaves. Communication port 0 of each pRing is reserved for bus communication. The MNR

architecture can be configured as a multiple-bussed system or single-bussed system. SimM can be extended to cover most possible communication topologies to meet future investigation requirements.

The pRing control programs define the activities of pRings. A pRing assembly language and corresponding machine code definition are included (Appendix A). The pRing control program to carry out their objective resides in the control memory of pRings. SimM can execute the control programs clock by clock, step by step or cycle by cycle as directed. This eases the debugging process. SimM also allows sharing pRing control programs to reduce the need for memory on the machine on which SimM is currently executing.

SimM can be divided into two disjointed processing phases: construction and simulation. In the construction phase, the simulation program reads architectural parameters provided by the user to construct the target MNR system. The flexibility of MNR architecture is represented by the availability of architectural parameters supported by the simulation program. Currently, SimM supports eleven architectural parameters plus topology assignments and pRing control program assignments.

The construction phase of SimM validates the MNR architecture design. In the simulation phase, the program simulates the activities of the MNR architecture, which, in turn, simulates ANN models. SimM executes pRing programs as the actual MNR system would do. Aside from providing ANN model results, SimM gives data such as elapsed simulation time and device utilization. Various factors can be derived from the data provided by the simulation program. The relationships among the subprocesses of the simulation program are shown in Fig. 23.

In this phase, the Global Control is constructed for a proposed MNR system with the architectural parameters given by the user. The Global Control instructs the PRING to configure itself according to pRing-related architectural parameters. The PRING then generates PEs along with their corresponding accumulator-memory and weight memory. The PRING also creates a PCB (pRing Control Block) for every pRing. The Program Loader loads the pRing control program for every pRing and then puts the starting address and length of the pRing control program into the PCB.

The Global Communication Control uses communication-related architectural

parameters and topology parameters to construct the communication channels of the proposed MNR system. Fig. 24 shows the construction phase of SimM.

The Monitor dominates the simulation process during the simulation phase. The user controls simulation by using the Monitor. Performance and utilization data are produced by the Monitor. Although there typically exists only a single copy of the PRING subprogram, Monitor calls PRING with different PCBs to create the illusion of more than one pRing in the system. For every clock period, Monitor preferably presents every PCB to PRING once. That is, each pRing executes one clock cycle of its pRing control program. This can be viewed as a time-sharing system in which every pRing has equal time slices. In this way, Monitor simulates an MIMD MNR system on an SISD (Single Instruction-stream Multiple Data-stream) machine. Fig. 25 shows the Simulation phase of the simulation program.

The SimM is composed of several functional units: *Global Control, Global Communication Control, Program Loader, PRING, HOST* and *Monitor*. Global Control sets up the simulation environment. Global Communication Control handles inter-ring communications. Program Loader loads the pRing control programs for every pRing. PRING executes pRing control programs. Monitor controls the simulation environment and acts as a bridge between the MNR and the outside world.

Global Control constructs the MNR system by setting up a PCB for every pRing. The contents of the PCBs are associated with the architectural parameters provided by the user (Appendix C). The PCBs keep the pRing processing status information during simulation. *Program Loader* loads the pRing control programs for every pRing. Program Loader then puts the starting address and length of the pRing control program in the PCB associated with that pRing. Fig. 26 shows the Global Control and the Program Loader of SimM.

The pRings are the core of the MNR architecture. As discussed previously massive parallelism proposed in this architecture comes from the parallelisms among PEs and among pRings. SimM treats each pRing as if it were an independent CPU executing its own control program. All the synchronous problems encountered will be resolved by the techniques used in solving communication and networking problems.

A pRing is composed of PEs, an I/F, a CM and a CU. The entire pRing operation is controlled by the CU, which actually comprises three major components: MCU (master

control unit), PCU (PE control unit) and ICU (interface control unit). Each of them serves a unique function within the pRing. The MCU fetches instructions and passes them to the PCU and the ICU. The PCU handles neuron arithmetics and the ICU settles communications. The PCU and the ICU are viewed as special-purpose co-processors attached to the MCU, which is the central processor. All processors execute their own classes of instructions simultaneously. There exist in MNR machine language special instructions to synchronize these processors. In simulating these activities, SimM is able to manipulate both the parallelisms among PEs, and among and within pRings, as well as the serialism within the MCU, the PCU and the ICU.

PRING executes the pRing control program as a real pRing would do. A pRing control program can contain three classes of instructions: local instructions, PE instructions and interface instructions. MCU executes local instructions which are mostly housekeeping or conditional program control transfers. PCU executes PE instructions which are mostly vectorized arithmetic instructions. ICU executes interface instructions which involve inter-pRing communication. Fig. 8 shows the pRing from a programmer's point of view.

In SimM, there typically exists only a single copy of PRING. SimM repeatedly reuses the only copy of PRING with different PCBs to simulate more than one pRing executing simultaneously. One can recognize this technique as fixed memory multiprogramming management; pRing is a piece of fixed memory, there are a lot of virtual pRings to be allocated to this pRing. The only difference from memory management is that the virtual pRing allocation is sequential in SimM. The same technique applies to the PEs, SimM has only a single copy of the PE. Through reusing the same piece of code, SimM creates an illusion that a lot more PEs exist on the system.

This way SimM can simulate various numbers of pRings and PEs without modification. The workloads of PCU, MCU and ICU are different from each other. It may be necessary to use different clock rates for each of them. SimM can execute different classes of instructions in different clock frequencies. To define system clock frequency, use the architectural parameter **CLOCK**. The device clock can defined as a multiple of the system clock. **PE_SPEED, CU_SPEED** and **IF_SPEED** define clock speeds for PCU, MCU and ICU respectively. Number of pRing is defined by the architectural parameter

NO_PRING.

Every pRing in SimM has a PCB associated with it. Each PCB contains pRing-related architectural parameters and the current pRing operating status. PRING is invoked by Monitor once per clock cycle. Once PRING is invoked, every pRing on the system executes a single clock cycle. The pRing invokes its components, namely, MCU, PCU and ICU, if the current clock cycle count is a multiple of the components' design speed. Fig. 27 shows the PRING module of SimM.

SimM assumes HOST is another pRing without PEs. A HOST control program contains only interface instructions and local instructions. Fig. 28 shows the HOST module of the simulation program.

The master control unit, *MCU*, is responsible for fetching instructions from the control memory and either executing them or dispatching them to other control units. The MCU also contains an ALU (Arithmetic-Logic unit) and a small set of registers, which are used for housekeeping in execution of the pRing control program. Fig. 7 shows the block diagram of the MCU.

A. Instruction fetcher and dispatcher.

Instruction fetching and dispatching time is intended to be fully overlapped with execution time, even if it turns out that the overlapping is not necessary according to the simulation results. MCU is tyically the only unit in pRing which has the privilege of accessing control memory. MCU fetches an instruction, then decides to which controller it should dispatch this instruction by judging the first and second most significant bits of the instruction. ICU and PCU each have a 4-word-length instruction buffer. The lengthy can be changed by recompiling SimM. MCU stops fetching the next instruction under two circumstances: the ICU (or PCU) instruction buffer is full and the next instruction is again an ICU (PCU) instruction, or the program counter reaches the end of the control program.

B. Housekeeping.

MCU keeps track of the status of ICU and PCU by examining the content of the status register. Both ICU and PCU will update the content of the status register according to their ongoing status, i.e., busy or not. MCU refers to the status register when MCU operates conditional program control transfers. Programmed Synchronization within pRing (i.e.,

among MCU, PCU and ICU) can be accomplished by busy-waiting for the units to be synchronized. Instructions below resynchronize MCU, ICU and PCU.

```
JPBSY   $   ;wait for PEs
JRBSY   $   ;wait for ICU receiving
JSBSY   $   ;wait for ICU sending
```

The status register contains normal ALU flags such as Zero, Sign, Carry, and control unit status flags like PBSY(PEs are busy), ISBY(Interface unit is busy in sending), IRBY(Interface unit is busy in receiving). Conditional control transfer instructions refer to the status register to achieve partial event-driven control.

Keeping all statistical information for the current pRing is part of MCU's job, though the real controller of PCBs in SimM is Monitor. Monitor collects information in the PCB to generate simulation statistics.

After fetching a PE instruction from the instruction buffer, PCU decodes the PE instruction and broadcasts PE microcodes to every PE in the pRing. In other words, PCU serves as a representative of PEs to manage control signals from MCU to PEs, or vice verse. MCU views all PEs together as a vectorized arithmetic processor. As do users of SimM. The number of PEs is defined by the architectural parameter **NO_PE_PRING.**

C. Processing element (PE)

Neuron activities occur within PEs. For each PE instruction issued by MCU, PCU transforms the instruction to process a data packet. A data packet contains a portion of the neuron data vector. The length of data packets is usually equal to the number of PEs in pRing, unless there exists fragmentation problems. As stated previously, a PE performs primitive ALU functions. Major functions of PE are addition and multiplication. These major functions are sufficient to perform neural computing. To keep PE as simple and flexible as possible, a bit serial ALU was selected for it although other components are available. As a result, MNR system performance will change as data precision changes. This is obvious, since PE needs a longer time to processes a data vector. This feature, which changes data precision not only affects data accuracy, but also processing speed. Thus, the

MNR has a trade-off of precision for speed. For SimM, it is nothing but a waste of time to simulate every step of the bit serial process. A decision was made to keep both the bit serial ALU feature and simulation speed. That is, to keep the timing needed for the bit serial ALU in PE, but do the actual arithmetic parallelly. This mechanism speeds up SimM's operation. The precision of data is limited to byte-boundaries in SimM although the timing of various lengths of precision are kept. The precision is set by the architectural parameter **ARITHMETIC_LEN**.

D. Weight memory

Weight memory holds a portion of the weight matrix of the simulated neural network. The size of weight memory is based upon the size of the weight matrix, the number of PEs in the system, and the number of neurons in the system. In any case the size of weight memory should be decided during pRing control program development. The size of the weight memory grows exponentially with the number of neurons in the fully connected network. Soon enough, weight memory allocation will substantially use up available main memory. So the size of the weight memory sets the limit for SimM. The architectural parameter **WMEM_SIZE** defines the weight memory size.

E. Accumulator-memory

Accumulator-memory stores the neurons' previous activation value and current partial sum. Accumulator-memory also serves as a scratchpad for each PE. If we view the PE as an ALU, accumulator-memory is the register file of this ALU. The **AMEM_SIZE** architectural parameter defines the size of the accumulator-memory.

F. Processing buffer

Each PE has three buffers, the transmitting buffer (T), the receiving buffer (R) and the processing buffer (P). Generally speaking, T and R both are controlled by the ICU instead of the PE. Every PE operation involves a P buffer, whether multiply-accumulate (XMAC), data movement (PUT and GET) or arithmetic (ADDA, SUBA . . .etc.). P buffers are circular buffers, so the content of the P buffer can be shifted to the next PE while the PE is processing the current content of the P buffer. Since we used a bit serial ALU for the PE, the processing of multiplication or addition is much slower than a buffer shift. That is, the time needed for buffer shifting can be fully overlapped with the time needed for

multiply-accumulate.

Dynamic memory allocation is used in SimM to implement memory and buffer. SimM implements the P buffer with linked list. Accumulator-memory and weight memory are implemented as linked arrays in SimM.

The interface control unit (ICU) is the I/O manager of the pRing. ICU controls the routine from and to the outside of the pRing. Sending and receiving data vectors always take place at the same time in each pRings. To resolve the sending and receiving bottleneck, ICU has different interface units for sending and receiving data vectors. The PE has different buffers to store incoming and outgoing data vectors. Both interface units access data vectors with pRing communication ports.

ICU instructs the interface unit to send out whatever is in the T buffer, whenever ICU encounters a SEND instruction in the pRing control program. Before the transmission begins, the communication link between pRings must be established. In SimM, ICU sends a sending request to the global communication control. The request will be granted only if both parties involved are ready to proceed.

The process for receiving a data vector is essential. ICU sends a receiving request to the global communication control. This request will be granted only if both ends of the communication link are ready. After the communication link is established, the data vector flows from the T buffer of the sending pRing to the R buffer of the receiving pRing.

Communication bandwidth is adjustable through architectural parameters. Hardware communication connections are determined by topology parameters. Communication links can be established between pRings only if there exists a hardware communication connection between the pRings.

The Global Communication Control resolves communication conflicts between pRings. In the construction phase, Global Communication Control builds the communication channels according to topology parameters and architectural parameters. In the simulation phase, Global Communication manipulates inter-pRing communication requests, either via a private channel or via the bus. Both communication conflicts and bus broadcasts are resolved by Global Communication. Fig. 29 shows the Global Communication Control of SimM.

A pRing involves both single-destination and multiple-destination communication. There

exists little difference from the pRing's point of view, since the ICU isolates the pRing from the outside world. But the Global Communication Control does distinguish multiple-destination communication requests from single-destination communication requests. For simplicity, **bus** communication for multiple-destinations communication and **point** communication for single-destination communication are used.

Bus communication involves a sending pRing and several receiving pRings. SimM uses a signed-up procedure to implement bus communication. Identities of pRings which want to receive data from a specific bus will be wait-listed by the Global Communication Control. The receiving pRings wait until a sending pRing wants to send data through that bus. The Global Communication Control informs those waiting pRings that they will receive data from the bus, while it tells the sending pRing to send out its data.

Global Communication Control typically allows only one sending pRing per bus or point communication link established. For point communication, there should be also be only one receiving pRing on the same communication link.

Global Communication Control monitors the current status of every communication connection. ICUs also report their current status to Global Communication Control, so Global Communication Control would have overview of the MNR system communication topology and status.

To prevent programmer's mistakes, Global Communication Control rejects communication requests involving illegal communication ports, nonexistent hardware connections or nonexistent buses. Global Communication Control rejects inconsistent topology definitions of topology parameters, i.e., hardware communication connections should contain both part of the communication link.

The major part of SimM is Monitor. Monitor consists of Monitor Interface, Error Handler and Monitor Control. Monitor Interface is the man-machine interface of the simulation program (see Appendix C for details of man-machine interface of SimM. Error Handler generates an alarms whenever an error occurs. The major function of Monitor Control is to manage PCBs. Monitor provides performance-related data and component utilization data from the contents of PCBs. Fig. 30 shows the component parts of Monitor.

The MNR architecture validation and evaluation are the major objectives of SimM. To

confirm an MNR architecture, the architecture is described in architectural parameters and topology parameters as stated above. SimM reports errors discovered during the construction phase. SimM subsequently reports inconsistency between pRing control programs and MNR architecture.

Communication connections are one of the major concerns in designing an MNR system. SimM is flexible enough to test all possible communication connections. SimM also has the option to determine communication channel bandwidth and clock timing of interface units.

Communication conflicts are very difficult to detect and resolve on paper. Complexity of the problems grows exponentially with the number of pRings. SimM can be used to resolve communication conflicts, whether the conflicts are caused by the topology or by the pRing control program.

The best way to debug a program is to run it. SimM simulates the MNR architecture to microcoded level, so SimM executes pRing programs that are compiled to MNR machine codes. SimM also provides a set of on-line commands to assist the debugging process.

SimM can be used as a utilization analysis tool to investigate both the architecture-dependent utilization and the ANN model dependent utilization. SimM provides at any time the utilizations of MCUs, PCUs and ICUs. By changing the architecture or execution speed of each device, the location of the execution bottleneck can be determined. This delivers information needed to design the most frequently utilized type of MNR hardware at minimum cost.

As stated previously, to change a simulated ANN model on SimM requires merely a change in pRing control programs. This way, different ANN models can be simulated by SimM on the same or even different architectures of MNR.

Architectural trade-offs of modules of MNR should be evaluated during the design phase of an MNR system. Those trade-offs, like fewer large pRings versus more small pRings, relative speeds of MCU, ICU and PCU, more point communication or a faster bus communication, serial ALU versus parallel ALU, etc., can be important factors in designing an MNR system. Those trade-offs can also be application-dependent. This is where an intensive simulation program is helpful. SimM is flexible enough to test possible combinations of architectural factors. SimM evaluates performance as part of architecture confirmation.

MNR architecture does not discriminate among ANN models. A different ANN model is merely a different set of pRing control programs for MNR. SimM can execute generally any pRing program assigned to it. To evaluate model-dependent performance, a model's pRing control programs need only be executed. The simulation status of SimM gives the performance of each individual model. From the information given by SimM, model dependent performance can be evaluated.

The performance and the properties of the MNR architecture, using SimM as a tool, are investigated, evaluated and discussed below. In discussion, the term 'Interconnections/second' refers to the number of multiply-accumulate operations that are performed in a second. DARPA has, in the past used speed (measured in interconnections per second) and capacity (measured in interconnects) as reference-variables. The same metrics are used in the following discussion. The capacity of the MNR architecture is limited only by the capacity of its weight memories; so the capacity of the MNR architecture is potentially infinite. Therefore, attention is focused on the performance of the MNR architecture in terms of speed and utilization. The inter-ring communication bandwidth, the number of PEs, the distribution of the PEs (i.e., the number of pRings on the bus), and the arithmetic precision (i.e., the number of bits needed to represent neuron-values and weights) are the major factors that affect the speed and utilization in the MNR architecture. The most advantageous trade-offs between these factors for the MNR architecture is illustrated below. The ANN model used in the tests is mainly the Hopfield model, though other models are also discussed.

Throughout the discussion, N represents the number of neurons in the system; K represents the number of PEs in the system; k represents the number of PEs in each pRing; M represents the number of pRings in the system; T stands for time, with subscript letters representing the specific operation (e.g. T_{mac} represents time needed for a bitwise multiply-accumulate operation); P stands for precision of arithmetic operations; I stands for interconnections and V represents system speed. Speed and capacity are two major factors that DARPA uses in evaluating ANN implementation tools. In DARPA's original speed/capacity plane (see Fig. 9), analog fully parallel architecture resides in the upper left corner of the plane, which indicates that it operates at very high speed, but with only limited capacity. Serial central processor occupies the lower half of the plane indicating that the architecture can be expanded

in capacity (by adding more memory) but with severe limits imposed on speed. The region between both architectures is where the MNR architecture fits. The speed of the MNR architecture can be expressed as:

$$V=\frac{I}{\frac{I}{K}\times T_{mac}\times P^2}=\frac{K}{T_{mac}\times P^2}=\frac{K^2}{T_{xmac}} \quad (1)$$

Equation (1) shows that the speed of the MNR architecture is independent of the size of the network, given that the neuron PE ratio remains constant. The equation also tells us that the system speed increases linearly with the number of PEs,and that the capacity (interconnections) increases linearly with the number of PEs (i.e., each PE comes with its own local memories). This indicates that both the speed and capacity of the MNR architecture grow linearly with the number of PEs. The performance and properties of the MNR architecture in simulating large scale models are derived from simulating and scaling smaller models.

Test#1 is designed to validate the speed equation (1). In Test#1, conditions are as follows:

1. N/K = 1
2. M = 1
3. T_{mac} = 200ns

With *P=8 bits*, the speed of the MNR architecture is

$$V=\frac{K}{1.28\times10^{-5}} \quad (2)$$

and if *P=16 bits*, the speed of the MNR architecture is

$$V=\frac{K}{5.12\times10^{-5}} \quad (3)$$

Test#1 is set up with a single pRing to eliminate the possible PE utilization loss due to the inter-ring communication. Fig. 31 shows the simulation results and the predictions from

computations: the speed of the MNR architecture increased linearly with the number of PEs in the system. The performance of the MNR architecture in terms of speed and capacity exactly resides within the middle region as predicted. The deviation in speed of the MNR architecture is caused by changing the degree of arithmetic precision. The speed of the MNR architecture is degraded by a factor of P^2 for different precisions, because of the bit-serial ALU design of PEs. But the linearity of the speed/capacity still holds for the various arithmetic precisions. The linearity in expansion provides the benefit of investigating the properties of larger model implementation by running smaller models on the MNR architecture. This can shorten the design and development time needed for large ANN model implementations.

The observed linearity also provides almost infinite expansion power to the MNR architecture in both speed and capacity, if the PEs are fully utilized. Unfortunately that is not always the case. Several factors such as communication bandwidth between pRings and neuron-PE ratio may reduce the PE utilization of the MNR architecture.

As stated previously, the performance of the MNR architecture in implementing large scale ANN models can be derived from the performance in simulating smaller models. To further investigate the properties of the MNR architecture, a 30-neuron fully-connected Hopfield model (interconnections) is tested.

The number of PEs (K) defines the processing power for the MNR architecture. The overall system performance (speed and capacity) grows linearly with the number of PEs. Test#2 is designed to verify the relationship between the MNR architecture performance and the number of PEs in the system. The capacity grows linearly with the number of PEs, since each PE carries its own weight memories. System speed (V) is considered in terms of the number of PEs in the system. The setup conditions for Test#2 are:

1. P = 8 bits
2. M = 1
3. N = 30 (I = 900)
4. T_{mac} = 200ns

In Test#2, K is the only variable. The changes in speed are caused solely by K. By changing K, the neuron-PE ratio (N/K) is also changed. System speed should decrease when

N/K increased, since each PE must serve more neurons in the system. Fig. 32 and Fig. 33 show the results of Test#2.

With reference to Fig. 32, the system speed of the MNR architecture grows linearly with the number of PEs. This is expressed by a speed function as:

$$V=CK \tag{4}$$

The equation (4) agrees with equation (1) if $C=\frac{1}{T_{mac}\times P^2}$, where both P and T_{mac} are constants. The relationship between system speed and neuron PE ratio is a linear degradation. Fig. 33 gives the degradation of speed when increasing the neuron PE ratio.

Now, the neuron-PE ratio is taken into consideration and the following equation is derived from Fig. 33:

$$V=Ctime\frac{K}{N} \tag{5}$$

Equation (5) reflects the degradation when increasing N/K. Comparing equation (5) with equation (1), a great similarity can be find. The only difference is caused by the fixed neuron PE ratio of Test#1. Thus a more general expression for speed for the MNR architecture can be derived from equations (1) and (5):

$$V=\frac{K}{N\times T_{mac}\times P^2} \tag{6}$$

Equation (6) reflects the degradation of increasing the neuron-PE ratio; i.e., each PE must serve more neurons. The degradations are also linear. Equation (6) can be explained as: the system speed of the MNR architecture grows linearly with the number of PEs in the system, is degraded linearly by the size of model, by the speed of basic multiply-accumulate

operations, and by the square of arithmetic precision.

In Figs. 32 and 33 the scales are logarithmic. These figures are redrawn and shown in Figs. 34 and 35 with linear scales on the abscissas. These figures provide more accurate readings.

The results of Test#2 meet the previously made assumptions: the processing power of the MNR architecture grows linearly with the number of PEs. The results have also shown, as already predicted, that speed is reduced when the neuron PE ratio is increased.

In the MNR architecture, the upper limit in the number of PEs is equal to the number of neurons in the system. If this limit is exceeded, the system performance will not increase when more PEs are introduced into the system. The MNR architecture operates at its maximum speed when the neuron PE ratio equals one. Thus, equation (6) is valid only for

Utilization, yet another important factor, which also affects the system speed. For the case where the neuron PE ratio is greater than one, the decrease in PE utilization will decrease the system speed. This problem is referred to as the 'fragmentation problem'. Equation (6) assumes that the PEs are fully utilized. Equation (7) covers the case where they are not fully utilized.

$$V = U_{PE} \times \frac{K}{N \times T_{mac} \times P^2} \qquad (7)$$

The distribution of the PEs over the pRings of the MNR architecture influences the number of circulation cycles within the pRings and the length of the inter-pRing communication data vector. Larger pRings will perform more efficiently if the communication channel cannot keep up with the processing speed of the PEs. Smaller pRings, on the other hand, will perform more efficiently if the bandwidth of the inter-pRing communication channel is large enough to cope with the fast processing PEs. Larger pRings will reduce the inter-pRing communication requirements, but would most likely suffer from the fragmentation problem. Smaller pRings will bring more flexibility to the MNR architecture, but will suffer from a slow communication channel. The various distributions of the PEs over the pRings of the MNR architecture, and their contributions to the MNR system performance is discussed

below. Test#3 is designed to investigate the effects of PE distribution in terms of speed and utilization. The setup conditions for Test#3 are:

1. P = 8 bits
2. K = 30
3. N = 30 (I = 900)
4. T_{mac} = 200ns

The number of pRings (*M*) is one of the variables in Test#3. The other is the inter-pRing communication bandwidth (*B*). The PE utilizations of a multi-ring MNR architecture are affected heavily by the communication bandwidth, and the PE utilization of the MNR architecture directly relates to the system speed. Figs. 36-38 show the results of Test#3.

Fig. 36 shows how the distribution of PEs affects the MNR performance under various communication bandwidths. The communication problem dominates the choice of the pRing size. In Fig. 36, the performance of the MNR architecture grows rapidly with the size of the ring if the inter-pRing communication is as slow as 10 Kbits/second. The growth is not so obvious, if the inter-pRing communication is as fast as 100 Mbits/second.

Figs. 37 and 38 show the changes in PCU utilization and ICU utilization when the number of PEs per pRing changes under various communication bandwidths. The ICU utilization decreases when the PCU utilization increases. By comparing Fig. 36 and Fig. 37, the performance of the MNR architecture grows with the PCU utilization. That is understandable, since the PEs offer the major processing power in the MNR architecture.

The distribution of PEs should provide an additional consideration in choosing the communication channels. And the communication bandwidth should provide an additional consideration in choosing between large pRings and small pRings. How will the communication bandwidth affect the performance of the MNR architecture under different PE distributions? To uncover the effects of communication bandwidth on the different distributions of PEs, the results of Test#3 are redrawn as shown in Figs. 39-41.

Fig. 39 shows the changes of speed in terms of communication bandwidth under different pRing sizes. The best performance of the MNR architecture happened in either a single pRing or when very fast communication channels exist.

The best PCU utilization and the worst ICU utilization, both of which reflect that most of

the system processing time is devoted to neural network processing and the least time to the communication overhead, happen in either a single pRing or very fast communication channels.

With reference to Figs. 39-41, the best performance attainable from a fixed number of PEs is to put them into a single pRing (i.e., $M=1$), or to distribute them equally over a number of pRings, given very fast inter-pRing communication channels.

The results of Test#3 suggest that larger pRings should be used. The larger the pRing, the higher utilization of the PEs. The larger the pRing, the faster the system operates. The best way is to put all the PEs in the same pRing. Unfortunately, the acceleration of performance stops at the point where the neuron PE ratio equals to one, because of the fragmentation problem stated above. pRing fragmentation is dealt with just memory fragmentation problems are dealt with in digital computers. A pRing is an allocatable unit within the MNR architecture. If k represents pRing size, i.e., $k=K/M$, then the average fragmentation loss would be approximately $k/2$. Another consideration in favor of smaller pRings is the advantage of modularity. Thus, if a fast communication channel exists between the pRings, then several small rings would give the best of both worlds.

The bandwidth of the inter-pRing communication channel determines controls the exchange rate of the data vector between pRings. If the communication bandwidth decreases, the utilization of PEs and the performance of the MNR architecture will also seriously degrade. Thus, the performance and the PCU utilization are both limited by the communication bandwidth. As above, the MNR architecture introduces three-way overlapping operations in the pRing, i.e., sending, receiving and processing. To evaluate the impact of the communication bandwidth an experiment is designed to test the various configurations. For example, Test#4 is designed to explore the role of the communication bandwidth in the MNR architecture. The setup parameters of Test#4 are:

1. P = 8 bits
2. M = 5
3. N = 30 (I = 900)
4. T_{mac} = 200ns

In Test#4, the number of PEs in the system, i.e., K is changed. These K PEs are equally

distributed over M pRings, so each pRing would contain k ($=K/M$) PEs. The communication bandwidth (B) is expected to be an important factor in determining the utilization of each device. The PEs are almost always waiting for communication, if B is too small (slow communication channels). The interface units are idling most of the time, if B is too large (fast communication channels). Since the PE's processing loads remain constant in terms of the number of multiply-accumulate operations, increased PE utilization will have the effect of improving system speed.

The behaviors of the MNR architecture under different communication bandwidths are shown in Fig. 42 and 43. The system speed of the MNR architecture grows as the communication bandwidth increases. Meanwhile, the PCU utilization grows rapidly with the communication bandwidth towards 100%. The ICU utilization drops, when communication bandwidth increases. The growth of the PCU utilization shoots up at a certain communication bandwidth. The utilization of ICU drops rapidly at the same communication bandwidth.

The system speed of the MNR architecture increases as the PCU utilization increases (equation (7)). The speed of the MNR architecture ceases to increase when the PCU utilization is limited. This result validates equation (7) and shows that whatever improves PCU utilization will also improve the MNR performance. The increase of the communication bandwidth does not yield much processing power to the MNR architecture, but does reduce the PCU idle time, if the PEs are idle due to the inter-pRing communication. When the PCU is almost fully utilized, the further improvement in the communication bandwidth does not have much effect on the MNR performance. The differences between Fig. 42 and Fig. 43 are caused by a different neuron PE ratio. The ICU utilization in Fig. 43 ($N/K=3$) drops sooner than that in Fig. 42 ($N/K=1$). This shows that the demands for communication decrease when the neuron PE ratio increases.

Inter-pRing communication time (T_{pRing}) is defined in terms of communication bandwidth (B), precision bits (P), neuron PE ratio (N/K), and pRing size ($k=K/M$). Equation (8) shows the time needed for a pRing to exchange a data vector with another pRing.

$$T_{pRing}=\frac{\frac{N}{K}\times K\times P}{B} \quad (8)$$

Since the pRing interface unit contains both sending and receiving devices, the sending and receiving data can take place at the same time. For M pRing, the communication time needed for a network cycle (assuming fully-connected network so every neuron needs all the others' values) should be $M-1$ times T_{pRing}. And so equation (8) can be rewritten as:

$$T_{comm}=T_{pRing}\times(M-1)=\frac{\frac{N}{M}\times P\times(M-1)}{B} \quad (9)$$

The utilization of ICUs regulates the contributions of interface units to system speed. T_{comm} defined in equation (9) assumes the ICUs are fully utilized which is definitely not the case. T_{comm} defines the system processing time spent on communications. ICU utilization defines the percentage of communication time in overall system processing time. Therefore, derivation of equation (10) from equation (9) is straightforward:

$$T_{sys}=\frac{\frac{N}{M}\times P\times(M-1)}{B\times U_{ICU}} \quad (10)$$

Test#5 is designed to investigate the communication bandwidth's influences on system speed, and to validate equation (10). Although it is desirable to have an unlimited communication bandwidth, most of the time a developer would have a fixed bandwidth communication link. Besides, the higher the communication bandwidth, the higher the cost and the lower the utilization of the communication channel. Since the number of PEs defines the processing power of the MNR architecture, it is important to find out how the speed relates to communication bandwidth and neuron-PE ratio. Test#5 is setup as folows:

1. P = 8 bits

2. M = 5

3. N = 30 (I = 900)

4. T_{mac} = 200ns

The number of PEs (K) and communication bandwidth (B) are the variables in Test#5. The conditions of Test#5 are essentially the same as in Test#4. The PEs are distributed equally over the M pRings. N/K is varied under various communication bandwidths in Test#5. Since N, M, and P are constants in Test#5, according to equation (10), the system speed is mostly affected by B and U_{ICU}.

With reference to Figs. 44 and 45, the need for communications decreases when the neuron-PE ratio increases. The communication bandwidth requirement is measured by ICU utilization. The ICU controls the pRing's interface unit. The ICU utilization then reflects the traffic of the communication channels.

The increase in PCU utilization when communication bandwidth grows is significant. The relationship of the PCU utilization to performance of the MNR architecture is different for different neuron-PE ratios.

Fig. 46 shows the variation in ICU utilization under various neuron-PE ratios. This shows that when the MNR architecture is simulating a larger ANN model (i.e., a large neuron-PE ratio), the communication bandwidth between pRings will not cause problems. The system processing speed does not increase linearly with PCU utilization ratio. This nonlinearity is caused by the communication problem. Equation (6) and equation (10) are combined to form a generalized MNR architecture processing time:

$$T_{MNR}=\max\left(\frac{\frac{N}{M}\times P}{B\times U_{ICU}},\frac{I\times N\times N\times T_{mac}\times P^2}{K\times U_{PCU}}\right) \tag{11}$$

$$V_{MNR}=\frac{I}{T_{MNR}} \tag{12}$$

Equation (11) assumes that the operations of PCU and ICU are fully overlapped. Whichever,

PCU or ICU, needs a longer time to complete a network cycle dominates the system speed in the MNR architecture. The PE processing speed dominates the system speed, when PCUs are almost fully utilized, and vice versa. Equation (12) gives the generalized system speed for the MNR architecture.

Bit-serial arithmetic processing ALU, residing in each PE, makes it possible to trade accuracy with speed and capacity. The arithmetic precision affects not only the arithmetic processing time in PEs but also the data exchange rate between pRings. Thus, the arithmetic precision influences the performance of the MNR architecture in two ways. First, it affects the PE processing time, i.e., if the precision is P bits, then a multiply-accumulate needs P^2 clock cycles to be completed. Second, it affects the communication time through communication channels. Test#6 is designed to find out what the impact of the arithmetic precision is on performance of the MNR architecture. The setup parameters of Test#6 are:

1. B = 1 Mbits/second
2. M = 5
3. N = 30 (I = 900)
4. T_{mac} = 200ns

Arithmetic precision (P) is changed to investigate the effects of P on system speed and utilizations. The PCU utilization is predicted to increase rapidly with the precision, since increasing precision puts more processing loads on PEs. Because the precision affects the processing loads within PEs, the neuron-PE ratio(N/K) is also changed to find out the properties of the MNR architecture in simulating larger models with different precision requirements.

Fig. 47 shows the MNR architecture's ability in handling various precisions by the bit-serial ALU with proper degradation. The MNR architecture downgraded gently when precision bits increased.

Fig. 48 shows the rapid increase of PCU utilization when the precision bits increase. The increases in PCU utilization does not lead to system speedup, since the overall network size (in terms of interconnections) does not increase.

The result of Test#6 shows the degradation in performance when arithmetic precision increased. The utilization of the PCU increases while the performance of the MNR

architecture decreases (Figs. 47 and 48). The ICU utilization decreases as predicted (Fig. 49). The reason for this is that the requirement of the PE processing time increases with P^2, while the communication time needed increases linearly with P. The effects the precision has on communication are covered by the increased PE processing need. From the overall speed point of view, decreasing the precision leads to speedup of the MNR system. This feature provides the benefits of trading precision with performance.

Thus far, an in depth investigation of the properties of the MNR implementation architecture has been presented. The flexibility and power of the SimM tool ease the testing tasks. Several equations have been derived from architectural analysis and verified by simulation. Values and trade-offs have been presented for evaluating the performance of the MNR architecture. It has been determined that:

$$T_{MNR}=\max\left(\frac{\frac{N}{M}\times P}{B\times U_{ICU}},\frac{I\times N\times T_{mac}\times P^2}{K\times U_{PCU}}\right)$$

$$V_{MNR}=\frac{I}{T_{MNR}}$$

These equations generalize the MNR architecture's performance in terms of architecture parameters such as I, N, K, M, B, T_{mac}, P, and of utilization figures.

Simulation results under various conditions and variables were also derived and presented. The results agree with the predictions given in equation (11) and (12). It was also shown that the MNR architecture can be expanded in both speed and capacity by adding more PEs to the system; thus demonstrates the MNR architecture's ability for modular expansion.

The MNR architecture is a modularly expandable system for large scale ANN implementation. Disregarding the PCU utilization loss due to the processing-control overhead, system speed and capacity of the MNR architecture increase linearly with the increased number of PEs. The MNR architecture satisfies cost-effectively the necessity for physical system implementations of the theoretical large-scale neural network models.

The problem with evaluating the MNR architecture is that the large number of parameters needed for evaluation rapidly increases the complexity of evaluating the architecture. Several tests of performance are designed and presented from which useful conclusions were drawn. First, in Test#1, the MNR architecture's position in the DARPA speed/capacity plane is presented (Fig. 9). The performance of the MNR architecture resides within the middle region as predicted. Both the speed and the capacity of the MNR architecture increase linearly with the number of PEs.

Test#2 shows that the MNR architecture's speed increases linearly with the number of PEs and decreases linearly with the neuron PE ratio. This feature gives the advantage of investigating the properties of a large model's implementation by running a smaller model on the MNR architecture.

Test#3 is designed to investigate the relationship between communication bandwidth and pRing size. The test shows that the best performance of the MNR architecture occurred when the PEs are nearly fully utilized. A low communication bandwidth in a multi-pRing system reduces the PEs' performance. Smaller pRings result in a lower system speed if given a slow communication channel. This suggests that a single large pRing may be preferred or a fast communication channel must be available.

Test#4 demonstrates the relationship between the communication bandwidth and the neuron PE ratio. From the results of Test#4, the demands for communication decrease when the neuron PE ratio increases. Thus, the communication bandwidth will not become a problem when the MNR architecture is used to simulate large ANN models (i.e.,).

The effect of the arithmetic precision was investigated with Test#5. Increasing the precisions puts more processing loads on the PEs (by P^2). Thus, PE utilization increases when the precision increases, but the increases in PE utilization make no contribution to system speed. This result shows that speed can be traded for accuracy.

In general, trade-offs in the MNR architecture may be used to increase PCU utilization. Increasing in PCU utilization will also increase the MNR system speed. These trade-offs have been discussed above for various conditions. It is suggested, by simulation, that certain parameters should be considered when designing a MNR implementation system, namely:

1. Neural-network size, the number of neurons (N) and the number of interconnections (I), because they define the processing load for the architecture.

2. PE processing speed (T_{mac}), because PEs are the major processing power of the architecture.

3. Arithmetic precision (P), because the primitive computation time (i.e., the time needed for a bitwise multiply-accumulate operation) of PEs increases by the square of P.

4. Number of PEs (K), because the speed and capacity increase linearly with the increased number of PEs, if they are fully utilized.

5. Inter-pRing communication bandwidth (B), because it limits the PE processing speed, if the communication channels are not fast enough. The utilization of PEs will decrease when the PEs need to wait for communication.

These parameters have a relation to system speed, which can be expressed as (11) and (12). The number of PEs determines the system's potential for speed. The utilization of the PCUs determines the percentage of this potential which is actually been put to use. From this, It can be determined whether the system is operating at its full potential for speed. The utilization of the ICUs determines the expandability of the system configuration, since low utilization of the ICUs indicates that the communication channels are under-used by the pRings. So, adding more pRings will speed up the system, not only because of the increased number of PEs, but also because of the even more timely delivery of data vectors. If the communication bandwidth is large enough, the increased number of pRings will not slow down the delivery of data vectors. Hence, the utilization of ICUs defines the communication bandwidth needed, which, in turn, determines the expandability of the system configuration.

From a designer's point of view, both high PCU utilization and high ICU utilization are desired. A high PCU utilization indicates that the system is operating at nearly its maximum speed. A high ICU utilization shows that the designer did not pay too much for the under-used communication channels. But as pointed out, a low ICU utilization indicates the expandability of the system. Thus, it is advisable to always seek high PCU utilization but do not insist on high ICU utilization.

Fig. 50 demonstrates another speed and capacity performance of the MNR architecture as estimated from simulations of Hopfield model and multi-layered ANN topologies for different

cases of parameterization of the MNR architecture. This too is superimposed on the speed-capacity map adapted from DARPA (Fig. 9). It is clearly shown that the MNR architecture occupies the difficult diagonal region of the map. Further analyzed, the MNR architecture offers additional cost-performance trade-offs, as illustrated in Fig. 51. The areas plotted on the map in this figure indicate the performance estimates for MNR implementations (of the Hopfield network) at different levels of cost for memory, processors and other system components. The areas marked reflect ranges of performance per $-cost as it is overrated by cost efficiencies from 10:1 to 100:1.

Fig. 52 shows the performance of a two pRing system processing a Hopfield ANN. The number of PEs per pRing is varied and the PE utilization and processing speed (in effective connections per second) is plotted. The processor utilization measures the percentage of time that the PCU is active instead of idling waiting for either an ICU synchronization point or waiting for the MCU to set up a new instruction block. From measurements taken in the lab, most of the PCU idle time is attributable to the MCU DMA set up overhead. Of course, PE utilization increases with the size of the network since the MCU overhead time is fixed and the PCU vector instructions times are strong functions of the number of PEs. The processing speed (in connections per second) increases linearly with the number of PEs which, to first order, is expected. Plotted on the same figure is the system's burst speed assuming no activation function processing or overhead.

Fig. 53 represents measurements taken when varying the number of pRings in the system. For this test, fully populated pRings (40 PEs each) were used. In agreement with the previous test and simulation results, both the PE utilization and processing speed increases with the number of processors in the system. The system's burst speed (assuming no activation function or communication overhead) is also ploted on the same figure. In obtaining the data for these charts, the mechanisms leading to the overhead were studied, and utilization and speed measurements taken. From this information, experimental data was extrapolated to a fully configured prototype system (i.e. with the maximum of 16 pRings on the bus). To achieve this level requires merely fabricating another ten pRings with no design modifications. Even for the modest sized prototype, over 10^7 interconnections per second are achieved.

A three-layered neural network with error backpropagation (BP) learning rule were also implemented. The measurements of the BP model were taken from temporal decomposition with 16-bit precision for weights and activations. A BP model includes a forward phase and a learning phase. Thus,a multi-layer feedforward model's measurement was taken from the forward phase of the BP implementation. Fig. 54 represents measurements taken when varying the number of pRings and the number of PE per pRings in the system. For this test, 1-5 pRings were used each with 8-40 PEs, so the measurements covers the performance figures ranging from 8 PEs to 200 PEs. Fig. 54 reassured the linear scalability of the MNR architecture. Whether the PEs are in the same pRings of not, The performance (interconnection/second) of the MNR architecture increased linearly with the number of PEs in the system.

Fig. 55 represents the performance of the MNR prototype hardware implementing a three-layered neural net with BP learning rule. Every learning cycle consists of a feedforward phase and a BP phase. The activation function of the BP model required a smooth function (e.g. a sigmoid function). The logistic sigmoid function was implemented using a quadratic Taylor series expansion in ten intervals. Since the table of constants for this approximation is stored in each PE, it is possible to have different activation function for PEs in the same pRing. The measurements in Figure 24 also covers the learning performance ranging from 8 PEs to 200 PEs. The learning performance, again, linearly increased with the number of PEs in the system.

The PEs in the MNR system are the source of the processing power. Thus the art of efficient programming in the MNR system is transformed to the subject of keeping as many PEs as busy as possible. Fig. 56 shows the PE utilization which represents the efficiency of MNR programming in accordance with the invention. The PEs in the system are kept at over 90% utilized as shown in Fig. 56. However, the utilization slightly increase as the size of the model increase. This accounts for the quadratic growth in processing requirement and the linear growth of communication requirements when the model grows. Thus, the MNR system will kept near 100% utilization (in terms of PE utilization) when implementing large models forwhich the architecture is suited.

Different ANN models are expressed by different pRing control programs for the MNR

architecture. Performance evaluation of different ANN models in the MNR architecture is accomplished essentially by determining the performance of the architecture when executing the corresponding pRing control programs. Utilization of the dynamic devices determine the major effect that the ANN models have on the performance of the MNR architecture.

Research efforts are needed to define the specific relationship between various ANN models and the utilization of the corresponding devices in MNR implementations. In other words, the representative instruction-mixes for different ANN models should be defined. The utilization of the corresponding devices, then, can be determined from these instruction-mixes. The performance of the MNR architecture in simulating various ANN models can then be investigated in more depth.

The investigation of the MNR architecture is currently based on a single-bus pRing structure. Other possible communication topologies are available. Other dynamic communication network topologies, such as multi-bus structures, are possible alternatives to the single-bus one.

The single-bussed MNR structure can be used, when the design allows any of the M pRings to connect to any other pRing. In this case, any pRing-pair can use the bus for communication. The MNR architecture also can support multi-destination communications, with which a pRing can send the same data vector to multiple pRing-destinations simultaneously.

When M, the number of pRings, is large, extremely fast busses are required, and special design and programming precautions must be taken to minimize the need for access to the bus. By providing several busses in the architecture, these precautions may be eased. Each pRing can connect to one or more of the available busses. The multi-bussed system not only reduces the communication load per bus but also provides a degree of fault tolerance.

Multiple busses also provide even more flexibility to the MNR architecture. The bussed pRing slab can then become an allocatable unit in an MNR workstation environment.

To obtain the most out of the MNR architecture, a high-level ANN specification language and the associated optimizing compiler is useful. The language preferably performs efficient assignment of PEs and communication scheduling. The details of the architecture are hidden from the programmer as much as possible. The optimizing compiler generates the required

pRing machine codes based on the current configuration of the system, since the MNR architecture is field-upgradable.

Occam has been proposed as a higher level language for ANN model implementation over transputers. ANSpec has been developed by SAIC to model massively parallel distributed systems which can also be used to specify and manipulate ANN models. NNDL(Neural Network Design Language) has been developed for processing vectorized data sets parallelly on neural networks. The MNR optimizing compiler is needed for the specification language of choice. The dynamic PE and pRing assignments and the communication scheduling according to the assignments affect the compiler.

The behavior of the digital computer is described by deterministic and precise languages. The digital computer generally can accept only clear and complete information. In contrast, ANNs can operate with fuzzy data to produce relatively reasonable results.

Fuzzy logic methods of data representation and processing can be applied to artificial neural networks. Fuzzy logic can be used to deal with uncertain information processed by ANNs.

Both neural network and fuzzy logic apply experimentally verified rules rather than algorithmic formulas to the reasoning problem. That is, inductive reasoning rather than deductive reasoning. This property provide a more direct emulation to the human brain. Therefore, incorporating neural network processing with fuzzy logic should supply a more representative solution to this world of fuzziness.

The pRings within the MNR architecture are essentially vectorized processors. The sum-of-product operations of neural networks are the basic matrix-computation operations. With the MIMD nature of the MNR architecture, there exist applications other than ANN implementations. The MNR architecture offers an excellent architecture for matrix operations. Applications that involve regular operations, like in matrix operations, will be suitable to execute on the MNR architecture. The data assignments in this kind of application should be handled very carefully. Studies should be made to find a general algorithm for the MNR architecture to be used on non-ANN applications.

Although the present invention has been described with reference to a preferred embodiment, the invention is not limited to the details thereof. Various modifications and

substitutions will occur to those of ordinary skill in the art, and all such modification and substitution are intended to fall within the sprit and scope of the invention as defined in the appended claims.

Appendix A Instruction Set and Machine Code Definition

MCU ASM Instructions							
ASM Instruction	type 1 1 5 4	len 1 1 3 2	OP_CODE 1 1 1 0 9 8 7	m 6	r2 5 4 3	r1 2 1 0	comments
mov dd, rr	1 0	0 1	0 0 0 0 0	0	rr	dd	
mov dd, @rr	1 0	0 1	0 0 0 0 0	1	rr	dd	
mov dd, #cnst	1 0	1 0	0 0 0 0 0	0	1 1 1	dd	
mov dd, @cnst	1 0	1 0	0 0 0 0 0	1	1 1 1	dd	
mov dd, rr	1 0	0 1	0 0 0 0 1	0	dd	rr	
mov @dd, rr	1 0	0 1	0 0 0 0 1	1	dd	rr	
mov @cnst, rr	1 0	1 0	0 0 0 0 1	1	1 1 1	rr	
illegal inst.	1 0	1 0	0 0 0 0 1	0	x x x	1 1 1	
add dd, rr	1 0	0 1	0 0 0 1 0	0	rr	dd	
add dd, cnst	1 0	1 0	0 0 0 1 0	0	1 1 1	dd	
add dd, @rr	1 0	0 1	0 0 0 1 0	1	rr	dd	
add dd, @cnst	1 0	1 0	0 0 0 1 0	1	1 1 1	dd	
sub dd, rr	1 0	0 1	0 0 1 0 0	0	rr	dd	
sub dd, cnst	1 0	1 0	0 0 1 0 0	0	1 1 1	dd	
sub dd, @rr	1 0	0 1	0 0 1 0 0	1	rr	dd	
sub dd, @cnst	1 0	1 0	0 0 1 0 0	1	1 1 1	dd	
cmp dd, rr	1 0	0 1	1 0 1 0 0	0	rr	dd *	* don't store result.
cmp dd, cnst	1 0	1 0	1 0 1 0 0	0	1 1 1	dd	
cmp dd, @rr	1 0	0 1	1 0 1 0 0	1	rr	dd	
cmp dd, @cnst	1 0	1 0	1 0 1 0 0	1	1 1 1	dd	
and dd, rr	1 0	0 1	0 0 1 1 0	0	rr	dd	
and dd, cnst	1 0	1 0	0 0 1 1 0	0	1 1 1	dd	
and dd, @rr	1 0	0 1	0 0 1 1 0	1	rr	dd	
and dd, @cnst	1 0	1 0	0 0 1 1 0	1	1 1 1	dd	
or dd, rr	1 0	0 1	0 0 1 1 1	0	rr	dd	
or dd, cnst	1 0	1 0	0 0 1 1 1	0	1 1 1	dd	
or dd, @rr	1 0	0 1	0 0 1 1 1	1	rr	dd	
or dd, @cnst	1 0	1 0	0 0 1 1 1	1	1 1 1	dd	
xor dd, rr	1 0	0 1	0 1 0 0 0	0	rr	dd	
xor dd, cnst	1 0	1 0	0 1 0 0 0	0	1 1 1	dd	
xor dd, @rr	1 0	0 1	0 1 0 0 0	1	rr	dd	
xor dd, @cnst	1 0	1 0	0 1 0 0 0	1	1 1 1	dd	
HALT	1 0	0 1	1 1 1 1 1	0	0 0 0	0 0 0	
SYNC	1 0	0 1	1 1 1 1 0	0	0 0 0	0 0 0	
jcc rr	1 0	0 1	0 1 0 1 0	0	cc	rr*	*rr 0..6 : jmp to addr (rr)
jcc xxxxx	1 0	1 0	0 1 0 1 0	0	cc	1 1 1*	**rr7 : jmp to addr xxxxx
jncc rr	1 0	0 1	0 1 0 1 0	1	cc	rr *	
jncc xxxxx	1 0	1 0	0 1 0 1 0	1	cc	1 1 1	
Ccc rr	1 0	0 1	0 1 1 1 0	0	cc	rr*	
Ccc xxxxx	1 0	1 0	0 1 1 1 0	0	cc	1 1 1*	
Cncc rr	1 0	0 1	0 1 1 1 0	1	cc	rr *	
Cncc xxxxx	1 0	1 0	0 1 1 1 0	1	cc	1 1 1	
rcc	1 0	0 1	0 1 1 0 0	0	cc	x x x	
rncc	1 0	0 1	0 1 1 0 1	0	cc	x x x	

cc	Z	0 0 0	ncc	NZ	0 0 0	
	P	0 0 1		NP(M)	0 0 1	
	C	0 1 0		NC	0 1 0	
	RBSY	0 1 1		NRBSY(RRDY)	0 1 1	
	SBSY	1 0 0		NSBSY(SRDY)	1 0 0	
	PBSY	1 0 1		NPBSY(PRDY)	1 0 1	
	TRUE	1 1 1		NTRUE(FALSE)	1 1 1	

PCU ASM Instructions

MAC	type 1 1 5 4	len 1 1 3 2	oph 1 1 1 0	s 9	rr 8 7 6	ff 5 4 3	dd 2 1 0	comments
xmac @dd,@rr,ff	0 1	0 1	0 0	1	rr	ff	dd	
mac @dd,@rr,ff	0 1	0 1	0 0	0	rr	ff	dd	
shift P	0 1	0 1	0 0	1	1 1 1	1 1 1	1 1 1	
nop	0 1	0 1	0 0	0	1 1 1	1 1 1	1 1 1	

P <-> A/W	type 1 1 5 4	len 1 1 3 2	oph 1 1 1 0	OP 9 8	p n 7	cc 6 5	a w 4	D 3	rr 2 1 0	comments
puta @rr	0 1	0 1	1 1	0 0	0	0 0	0	0	rr	D=0:
geta @rr	0 1	0 1	1 1	0 0	0	0 0	0	1	rr	P->A/W
putw @rr	0 1	0 1	1 1	0 0	0	0 0	1	0	rr	D=1:
getw @rr	0 1	0 1	1 1	0 0	0	0 0	1	1	rr	P<-A/W
putacc @rr	0 1	0 1	1 1	0 0	0	cc	0	0	rr	
getacc @rr	0 1	0 1	1 1	0 0	0	cc	0	1	rr	
putwcc @rr	0 1	0 1	1 1	0 0	0	cc	1	0	rr	cc cond
getwcc @rr	0 1	0 1	1 1	0 0	0	cc	1	1	rr	0 no
putancc @rr	0 1	0 1	1 1	0 0	1	cc	0	0	rr	1 Z
getancc @rr	0 1	0 1	1 1	0 0	1	cc	0	1	rr	2 C
putwncc @rr	0 1	0 1	1 1	0 0	1	cc	1	0	rr	3 S
getwncc @rr	0 1	0 1	1 1	0 0	1	cc	1	1	rr	

ALU	type 1 1 5 4	len 1 1 3 2	oph 1 1 1 0	OP 9 8 7 6 5 4 3	rr 2 1 0	comments
adda @rr	0 1	0 1	0 1	0 0 0 0 0 0 0	rr	A[d[rr]] += P
suba @rr	0 1	0 1	0 1	0 0 0 0 0 1 0	rr	A[d[rr]] -= P
cmpa @rr	0 1	0 1	0 1	0 0 0 0 0 1 1	rr	
anda @rr	0 1	0 1	0 1	0 0 0 0 1 0 0	rr	A[d[rr]] &= P
ora @rr	0 1	0 1	0 1	0 0 0 0 1 0 1	rr	A[d[rr]] |= P
xora @rr	0 1	0 1	0 1	0 0 0 0 1 1 0	rr	A[d[rr]] ^= P

Buffer	type 1 1 5 4	len 1 1 3 2	oph 1 1 1 0	OP 9 8 7 6 5 4	bufr 3 2	bufd 1 0	comments
mov bd, br	0 1	0 1	1 0	0 0 0 0 0 0	br*	bd*	*P 0
xchg bd, br	0 1	0 1	1 0	0 0 0 0 0 1	br*	bd*	R 1
							T 2

ICU ASM Instructions					
	type 1 1 5 4	len 1 1 3 2	OP_CODE 1 1 1 0 9 8 7 6 5 4	ports 3 2 1 0	comments
send port	1 1	0 1	x x x x 0 0 0 0	port	x: don't care.
rcv port	0 0	0 1	x x x x 0 0 0 1	port	

Appendix B Simulated ANN pRing Program Listing

Program for a 30 neurons hopfield net. Program shows program for pRing 1 in a 3 pRing, 5 PE-per-pRing MNR system.

```
0000                    cpu     "pring.tbl"
0000                    hof     "int8"
0000 =      bus:        equ     0
0002 =      right:      equ     2
0005 =      k:          equ     5           ;pe/pring
003D =      zero:       equ     61          ;address of
003E =      thrsh:      equ     62          ;constants in
003F =      act1:       equ     63          ;weight
0040 =      act2:       equ     64          ;memory

0000 A03D0064           mov     r5,#100     ;for 10 network cycles
0004        lp1:
0004 A038003D           mov     r0,#zero    ;..init accum (1,2) to          0
0008 5C18               getw    @r0
000A A0380001           mov     r0,#1
000E 5C00               puta    @r0
0010 A0380002           mov     r0,#2
0014 5C00               puta    @r0
0016 A03B0001           mov     r3,#1       ;..set xmac offset = 1
001A A03A0001           mov     r2,#1       ;..wbase = 1
001E A0380003           mov     r0,#3       ;..p = 3
0022 5C08               geta    @r0
0024 A0390001           mov     r1,#1       ;..xmac 1,wbase,1
0028 52D1               xmac    @r1,@r2,r3
002A A13A0005           add     r2,#k       ;..wbase = wbase + k;
002E A0390002           mov     r1,#2       ;..set up for next xmac
0032 A52F0032           jpbsy   $           ;..wait             PE
0036 52D1               xmac    @r1,@r2,r3;..xmac 2,wbase,1
0038 A13A0005           add     r2,#k       ;..wbase = wbase + k
003C A52F003C           jpbsy   $           ;..wait             PE
0040 5802               mov     t,p         ;..t=p
0042 A0380004           mov     r0,#4       ;..p=4
0046 5C08               geta    @r0
0048 A03C0003           mov     r4,#3       ;..for 3 iterations {
004C        lp2:
004C A0390001           mov     r1,#1       ;....xmac 1,wbase,1
0050 52D1               xmac    @r1,@r2,r3
0052 D002               send    right       ;....send right
0054 1010               rcv     bus         ;....rcv bus
0056 A13A0005           add     r2,#k       ;....wbase = wbase + k
005A A0390002           mov     r1,#2       ;....set up for     next xmac
005E A52F005E           jpbsy   $           ;....wait PE
0062 52D1               xmac    @r1,@r2,r3;....xmac 2,wbase,1
0064 A13A0005           add     r2,#k       ;....wbase = wbase + k
0068 A51F0068           jrbsy   $           ;....wait rcv,snd,pe
006C A527006C           jsbsy   $
0070 A52F0070           jpbsy   $
0074 5802               mov     t,p         ;....t = p
0076 5804               mov     p,r         ;....p = r
0078 A13CFFFF           add     r4,#-1      ;..}
007C A547004C           jnz     lp2
0080 A0390001           mov     r1,#1       ;..xmac             1,wbase,1
0084 52D1               xmac    @r1,@r2,r3
0086 D002               send    right       ;..send             right
0088 1010               rcv     bus         ;..rcv bus
008A A13A0005           add     r2,#k       ;..wbase = wbase + k
008E A0390002           mov     r1,#2       ;..set up next xmac
0092 A52F0092           jpbsy   $           ;..wait             pe
0096 52D1               xmac    @r1,@r2,r3;..xmac 2,wbase
0098 A13A0005           add     r2,#k       ;..wbase = wbase + k
009C A0390001           mov     r1,#1       ;..set up next xmac
00A0 A51F00A0           jrbsy   $           ;..wait             rcv,pe
00A4 A52F00A4           jpbsy   $
00A8 5804               mov     p,r         ;..p = r
00AA 52D1               xmac    @r1,@r2,r3;..xmac 1,wbase,1
00AC A13A0005           add     r2,#k       ;..wbase = wbase + k
00B0 A0390002           mov     r1,#2       ;..set up next xmac
00B4 A52F00B4           jpbsy   $           ;..wait             pe
00B8 52D1               xmac    @r1,@r2,r3;..xmac 2,wbase,1
00BA A0390001           mov     r1,#1       ;..Compute activation (1 -> 3) and (2 -> 4);
00BE A03A0003           mov     r2,#3
00C2 A73F00E0           call    active
00C6 A0390002           mov     r1,#2       ;..Compute activation (1 -> 3) and (2 -> 4);
00CA A03A0004           mov     r2,#4
00CE A73F00E0           call    active
00D2 9F00               SYNC
00D4 A13DFFFF           add     r5,#-1      ;}
00D8 A5470004           jnz     lp1
00DC A53F00FC           JMP     STOP        ;ALL DONE

        ;activation computation. binary          output
        ;r1 = @summation, r2 =   @activation
```

```
00E0          active:
00E0 A038003E      mov     r0,#thrsh     ;fetch threshold from          weight mem
00E4 5C18          getw    @r0
00E6 5411          suba    @r1           ;acc - threshold
00E8 A038003F      mov     r0,#act1      ;get low value of activation
00EC 5C18          getw    @r0
00EE A0380040      mov     r0,#act2      ;set up          to get high value
00F2 5C78          gtwp    @r0           ;get high value  if acc > thresh
00F4 5C02          puta    @r2           ;put away value
00F6 A52F00F6      jpbsy   $
00FA 9638          ret

00FC 9F80     STOP: HALT
0000           END
```

Appendix C SimM User's Manual

C.1 Introduction to SimM

SimM (for "*Sim*ulation of *M*NR") is intended to represent the MNR architecture in software modules, to provide an experimental environment for architectural trade-off confirmation and to allow ANN model dependent performance analysis. SimM also will be used as a MNR architecture development tool to assist in communication conflict resolution, to debug pRing programs, to analyze hardware utilization and to experiment with various hardware configurations. SimM can also serve as an ANN model development tool to confirm the predicted activities of a proposed new ANN model.

To accomplish the objectives stated above, SimM should be able to restructure itself through changes in parameters. These parameters include *architectural parameters, topology parameters* and *pRing control programs*.

SimM also provides a set of on-line commands. One can operate SimM as if operating a real MNR machine. This tool allows the developer to examine and change the value of accumulator-memory and weight memory, to monitor the changing ANN states at various times and to check current utilization of devices. SimM offers all the features needed for developer, and more. With SimM, one can test a new ANN model on the MNR system as well as test a new MNR design with an existing ANN model.

C.2 SimM command line parameters

At the DOS prompt (i.e., C:>) type **SIMM** without any command line arguments. SimM will response with:

USAGE:

SIMM <architecture> <topology> <program> <neuron map> <weight matrix>

SimM is telling you that some important parameters are necessary for processing. We will discuss these parameters in this chapter. After finishing this chapter, you will be able to create a MNR architecture with those parameters on your own. The architecture file contains definitions of the MNR hardware architecture. The topology definition file has the definitions of pRing connections. The program file defines the pRing control program. The neuron map file tell SimM where neurons are and where to show them. The weight matrix file holds the initial value of weight memory. With these parameters, SimM builds a MNR system environment to be used.

C.2.1 Architecture file

The architecture file contains parameters needed to create pRings. The parameters are entered as a keyword followed by an integer. Table 1 shows the currently available parameters and their meanings.

A sample architecture file for a 3-pRing, 5-PE-per-pRing MNR system is presented in Figure C.1. The MNR system has 3 pRings. Each pRing contains 5 PEs. Each PE has 61 bytes of weight memory and 16 bytes of accumulator-memory. All PEs employ 16 bit arithmetic operations. The relative system clock runs at 20 Mhz(50 ns clock cycle). PEs of the MNR system run at 20 Mhz while the MCU runs at 4 Mhz, and the Interface(IF) runs at 2 Mhz. Every pRing in the system has 5 communication ports each with a communication bandwidth of 16 Mbits/second.

KEYWORD	MEANING
NO_PRING	number of pRings on system.
NO_PE_PRING	number of PEs within each pRing.
WMEM_SIZE	size of weight memory within each PE (bytes).
AMEM_SIZE	size of accumulator-memory within each PE (bytes).
COMM_PORT	number of communication ports for a pRing.
CHANNEL_BANDWIDTH	channel bandwidth of communication lines (bits/IF_SPEED).
ARITHMETIC_LEN	arithmetic precision (bits) of PEs.
CLOCK	relative system clock cycle time (*ns*).
CU_SPEED	pRing control unit (MCU) speed according to relative system clock.
PE_SPEED	PE speed according to relative system clock.
IF_SPEED	Interface unit (IF) speed according to relative system clock.

Table 1. Architectural parameters

```
NO_PRING 3
PE_PRING 5
ARITHMETIC_LEN 16
CHANNEL_BANDWIDTH 8
COMM_PORT 5
CLOCK 50
PE_SPEED 1
CU_SPEED 5
IF_SPEED 10
WMEM_SIZE 61
AMEM_SIZE 16
```

Figure C.1 Sample SimM architecture file

C.2.2 Topology definition file

The topology file starts with the keyword **TOPOLOGY**, followed by pRing connection definitions. Every row of data begins with a pRing identity (ID), followed by the bus connection, then the communication ports. pRing ids range between 1 and the maximum number of pRings specified in the architecture file. The row of data whose pRing ID is 0 defines the bus identity of this MNR system. All bus connections should refer to the bus identities defined in row 0. Communication ports should contain either a pRing identity or -1(no connection). A legal pRing identity in a pRing communication port means that a communication link is established between them. SimM will connect the current pRing with the pRing whose identity shown in the communication port indicated. pRings communicate by way of communication ports, so the topology definition should always remain consistent, i.e., connection definition should by specified by both parties of a communication link. If pRing 1 has a connection to pRing 2 through one of pRing 1's communication ports, then pRing 2 should have a connection to pRing 1 via one of its communication ports also.

SimM will automatically reject illegal pRing and bus designators. SimM also checks topology consistency, and requests the error, if there is one, be fixed before any further processing. Figure C.2 is a sample topology definition file. In Figure C.2, we defined a single bussed MNR with three pRings. The connections in Figure C.3 show the topology resulting from the definition in Figure C.2.

```
TOPOLOGY
0 0  1 -1 -1 -1 -1
1 1 -1  2 -1 -1 -1
2 1  1  3 -1 -1 -1
3 1  2 -1 -1 -1 -1
```

Figure C.2 Sample topology definition file

C.2.3 Program file

Every pRing has its own special control program to carry out the tasks designated to it. In real system the control programs are provided by host computer during system initialization. During simulation, you tell SimM where to find the control program for every pRing. The control program should be compiled to MNR machine language in standard Intel Hex format. You put the information about control programs in a program file to tell SimM where to find them. The program file begins with the keyword **PROGRAM** and end with the keyword **PROGRAM_END**. Every individual pRing control program filename has it own line in the program file between the keyword **PRING_ID** and the keyword **END**. The number following **PRING_ID** is, obviously, the pRing ID to which the pRing control program belongs. Different pRings could use the same pRing control program by putting more than one **PRING_ID** line before the **END** line. Figure C.4 shows the program file for a 3-pRing MNR. pRing 0 is designated to be the host computer or the interface to the host computer. There is no control program for pRing 0 since at present, we don't consider the interfacing problem with the host computer. Eventually we will take that into consideration, at which point all you will need to do is to put the host (or host interface) control program in the program file.

```
PROGRAM
PRING_ID 0
END
PRING_ID 1
hopa1.hex
END
PRING_ID 2
hopa2.hex
END
PRING_ID 3
hopa3.hex
END
PROGRAM_END
```

Figure C.4 Program file

C.2.4 Neuron map file

The neuron map file is the way to tell SimM where the neuron activation values are and where on screen you want to see them. Neuron map files have two portions: definition and neuron map. The definition has only a single line in the neuron map file--the first line. The definition specifies the number of neurons in the system and the threshold to display as *active* or *inactive*. A neuron map specifies a specific accumulator-memory location as a specific neuron. Every line of a neuron map specifies a single neuron by giving the neuron ID, pRing ID, PE ID, accumulator-memory address, page of screen, and the row and column on screen.

Figure C.5 shows a neuron map file with 30 neurons with a threshold of 0. The activation value for neuron 0 is stored in pRing 1, PE 0, accumulator-memory address 3. Neuron 0 is to be shown on the first page of the screen at row 2, column 4. The display screen of SimM measures 48 by 16. That make maximum number of neurons in a single page limits to 768, but SimM allows multiple page display to overcome this limitation. Figure C.5 shows the neuron definitions in a neuron map file.

```
30 0
0  1 0 3 1 2 4
1  1 1 3 1 2 6
2  1 2 3 1 2 8
3  1 3 3 1 2 10
4  1 4 3 1 2 12
5  1 0 4 1 3 4
6  1 1 4 1 3 6
7  1 2 4 1 3 8
8  1 3 4 1 3 10
9  1 4 4 1 3 12
10 2 0 3 1 5 4
11 2 1 3 1 5 6
12 2 2 3 1 5 8
13 2 3 3 1 5 10
14 2 4 3 1 5 12
15 2 0 4 1 6 4
16 2 1 4 1 6 6
17 2 2 4 1 6 8
18 2 3 4 1 6 10
19 2 4 4 1 6 12
20 3 0 3 1 7 4
21 3 1 3 1 7 6
22 3 2 3 1 7 8
23 3 3 3 1 7 10
24 3 4 3 1 7 12
25 3 0 4 1 8 4
26 3 1 4 1 8 6
27 3 2 4 1 8 8
28 3 3 4 1 8 10
29 3 4 4 1 8 12
```

Figure C.5 Contents of a neuron map file

C.2.5 Weight matrix file

Although you can access every weight memory location in SimM, it is a time consuming process to assign every weight memory an initial value. SimM can invoked with a weight matrix as a parameter. The weights should be organized to match the requirement embedded in the pRing control program. SimM assumes the weight matrix file, whose filename appears as parameter when SimM invoked, is correctly organized as required.

The requirements for a weight matrix file are:

1. The weights should be organized by pRing numbers.
2. The number of columns in a line should equal the number of PEs in the pRing.
3. The number of lines in a weight matrix file should equal the weight memory size times the number of pRings.
4. The first line of a weight matrix file is assigned to first weight memory location of the first pRing, i.e., column 1 goes to PE 0, column 2 goes to PE 1. . . etc.

Conversion program to convert a weight matrix to the weight matrix file format of SimM is also available as a supplement to SimM. WCNVT takes a conventional weight matrix as input and outputs a SimM format weight matrix file. Since SimM reads standard ASCII files, you may want to type the initial value of weight memory in SimM format to a file. SimM can read that file as a weight matrix later.

Figure C.6 is part of a weight matrix file for a 3-pRing, 5-PE-per-pRing Hopfield net. Two patterns are stored by this Hopfield net.

```
0 0 0 0 0
2 0 2 2 0
0 0 0 2 0
0 2 0 0 0
0 2 2 0 2
0 0 2 0 0
0 -2 2 0 -2
-2 -2 0 0 -2
-2 0 0 -2 -2
-2 0 2 -2 0

    ...
    ...

-2 2 -2 0 0
0 0 0 0 2
0 0 2 0 0
0 -2 0 2 -2
-2 -2 -2 0 0
0 0 -2 -2 -2
-2 2 0 -2 0
```

Figure C.6 Part of sample weight matrix file

C.3 SimM on-line commands

After SimM is invoked, SimM sets up the MNR configuration according to your parameter files. If successful, SimM will show you a monitor screen like Figure C.7 (the italics do not appear on screen.) Otherwise SimM responds with an error message. The screen is divided into 5 portions *current command, error message, information window, simulation*

state, and *command* menu. All but the command menu are message areas showing the current status of the simulation. We will discuss commands in detail later. Table 2. Shows the contents and meanings of these message areas although they are self explanatory.

Area	Contents
current command	Shows the latest command issued
error message	Shows current error encountered
information window	Shows current activation state of neurons defined in neuron map file and other simulation informations.
simulation state	shows current simulation state like, clocks, times, network cycles and current page of neuron layouts.

Table 2. Message areas of SimM monitor

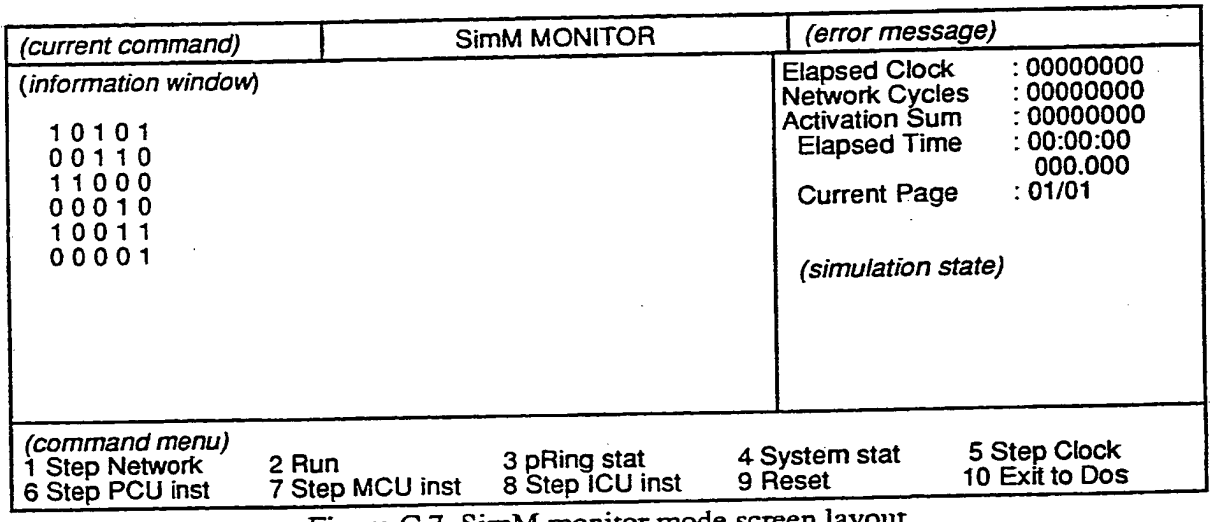


Figure C.7 SimM monitor mode screen layout

Figure C.8 shows all on-line commands provided by SimM. Most of SimM's on-line commands are shown on the *command menu*. Very few commands are not shown on menu including, *toggle screen display*(**Alt-F1**) under monitor mode, *modify accumulator-memory globally* (**Alt-F6**) and *modify weight memory globally*(**Alt-F7**) under pRing mode. Also **PgUp** and **PgDn** will change pages in both monitor and pRing modes. Figure C.9 shows the MNR system information provided by SimM (i.e., the user has pressed **F4**).

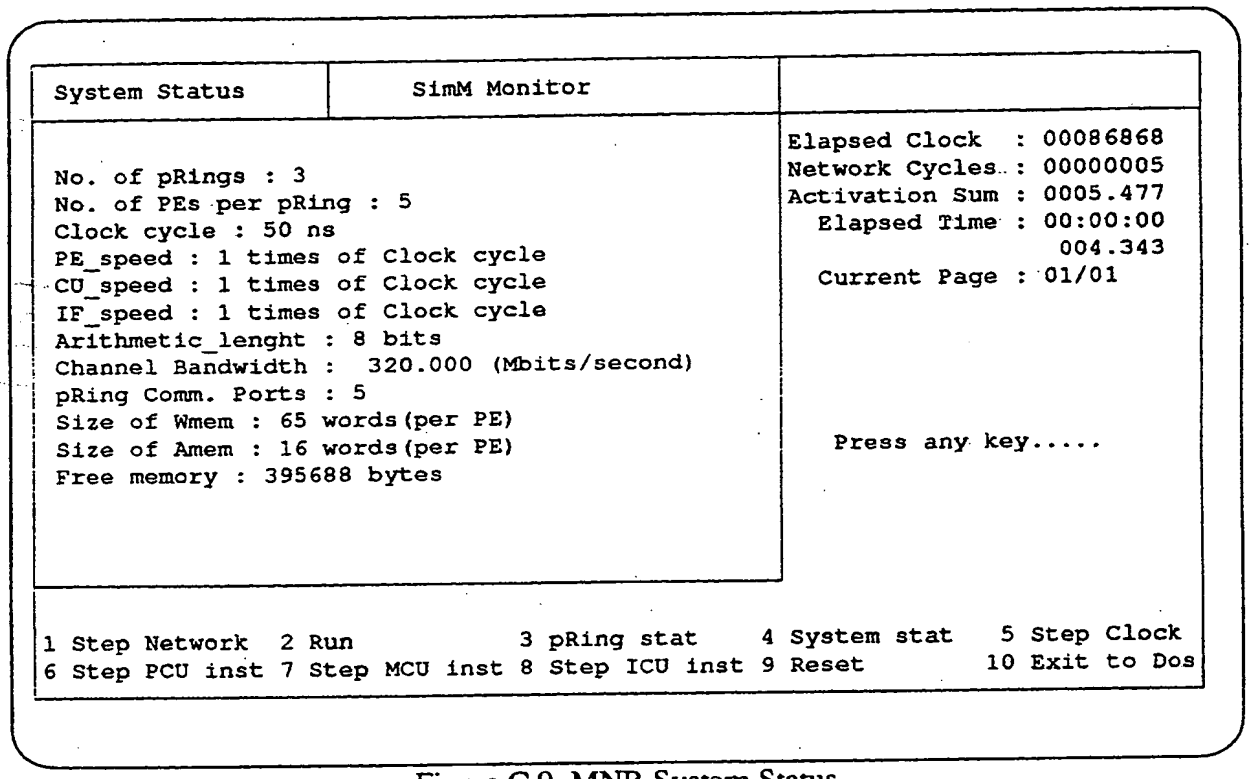


Figure C.9 MNR System Status

Some commands need additional information. When you request SimM to *step over MCU instruction*(**F7**) in the monitor mode, SimM will ask for the ID of the indexed pRing. The indexed pRing serves as a relative check point for the other pRings. When you ask SimM to step over the MCU instruction of a specific pRing, SimM will treat the pRing as the indexed pRing and execute all the pRing programs until the indexed pRing completes a MCU instruction. Or when you tell SimM that you want to see what's in accumulator-memory(**F2**). SimM will ask you to specify which PE in the current pRing you want to see. Of course, SimM will ask you the address of accumulator-memory or weight memory when you tell SimM you want to change their values by pressing **F6** or **F7** in the pRing mode. Global changes (**Alt-F6, Alt-F7**) affect only the current pRing.

C.3.1 Monitor mode commands

Screen display is relatively slow since we used the ANSI driver to maintain portability. We suggest turning the screen display off while running large models to speed up the simulation process. Also, you needn't enter the indexed pRing ID every time when you use the step command to debug pRing control program because SimM automatically repeats your last command when you hit **enter** on the keyboard. The following is a command description for all commands in the monitor mode.

Keystroke	Command	Action of SimM
F1	Step network	Step over ANN network cycles. ANN cycles are specified by the user in the pRing control program.
F2	Run	Execute pRing control programs until all pRings finish their control program. Refresh screen whenever an ANN cycle is finished even if screen display is turned off.
F3	pRing status	Switch to pRing mode. SimM will ask for the pRing ID.
F4	System status	Shows current MNR configuration.
F5	Step clock	Execute a clock of control program for every pRing on system.
F6	Step PCU instruction	Execute until the indexed pRing finishes a PE instruction. SimM will ask for the indexed pRing ID.
F7	Step MCU instruction	Execute until the indexed pRing finishes a MCU instruction. SimM will ask for the indexed pRing ID.
F8	Step ICU instruction	Execute until the indexed pRing finishes an interface-(IF) instruction. SimM will ask for the indexed pRing ID.
F9	System RESET	System reset. SimM will read the weight matrix file to initialize weight memory. SimM will also randomly assign initial values to the accumulate memories in which the neuron activation values store.
F10	Exit to DOS	Confirm if you really want to exit. If you do, free all memories and return to DOS.
Alt-F1	Toggle screen display	Toggle screen display.

C.3.2 pRing mode commands

Pressing **F3**(*pRing status*) in the monitor mode will bring you into the pRing mode. After answering the **"pRing ID?"** question, SimM changes the command menu to the pRing mode. Figure C.10 shows the pRing mode screen layout.

The commands available in the pRing mode and their descriptions follow.

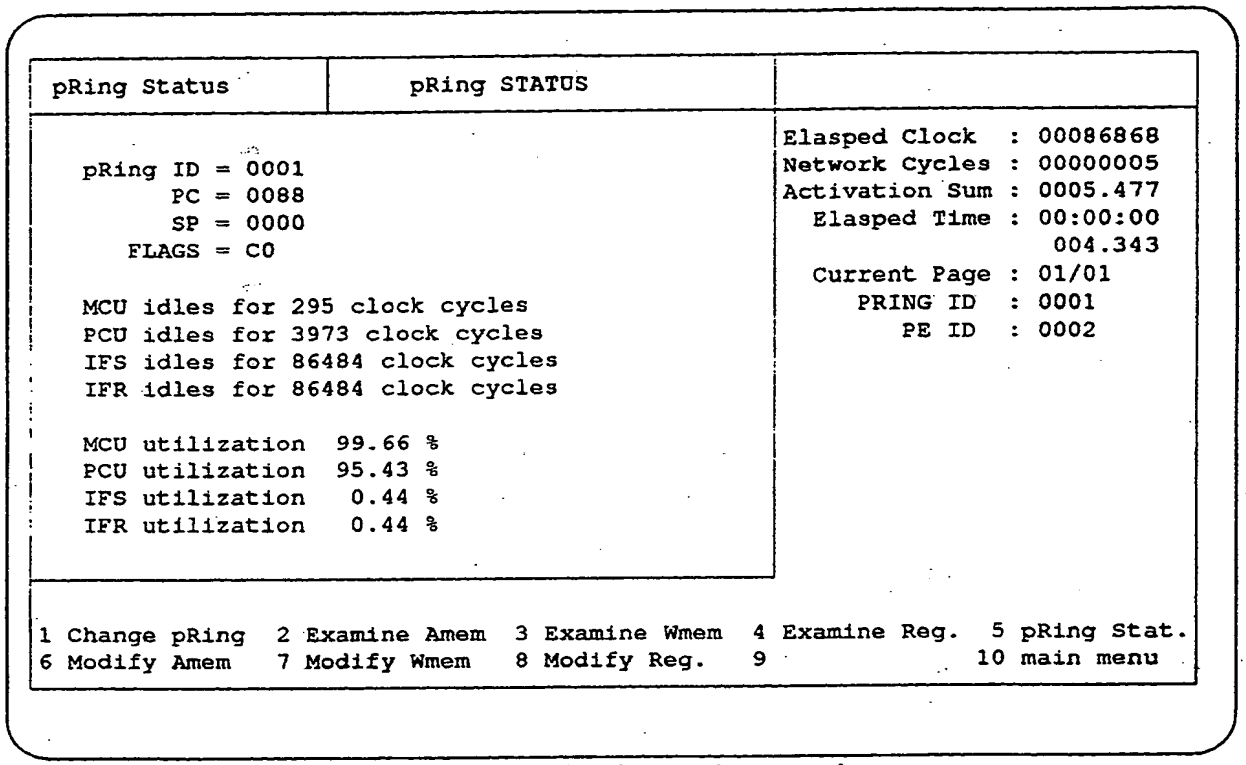


Figure C.10 SimM pRing mode screen layout

Keystroke	Command	Action of SimM
F1	Change pRing	Change current pRing. SimM will ask which pRing will be the current pRing.
F2	Examine Accumulator-memory	Check accumulator-memory of a specific PE within current pRing. SimM will ask you for PE to be checked and will then set the PE to be current.
F3	Examine Weight memory	Check weight memory of a specific PE within current pRing. SimM will ask you for PE to be checked and will then set the PE to be current.
F4	Examine pRing registers	Check pRing registers, including PC, SP, Flags and all 7 common registers.
F5	pRing status	Check current pRing status, including hardware utilization.
F6	Modify accumulator-memory	Modify accumulator-memory of current pRing and PE. SimM asks for address and new value.
Alt-F6	Modify accumulator-memory globally	Modify accumulator-memory for every PE of current pRing. SimM asks for address and new value.
F7	Modify weight memory	Modify weight memory of current pRing and PE. SimM asks for address and new value.
Alt-F7	Modify weight memory globally	Modify weight memory for every PE of current pRing. SimM asks for address and new value.

F8	Modify pRing register	Modify register value for current pRing. SimM asks for register number and new value.
F10	Return to monitor mode	Return to monitor mode.

C.4 Limitation of SimM

There are some constraints which apply to SimM. One of them is the limitation of memory. SimM runs under MS-DOS(PC-DOS) so SimM can't access more than 640 Kbyte of main memory although your machine may have several Mbytes. That limits SimM to only 200 K connections, i.e., 440 neurons fully connected. Speed is the other drawback of SimM. It runs relatively slow as an ANN model simulator. But our goal in designing SimM was to implement architectural simulation of the MNR architecture. Speed was not our major concern. Also, portability was a major concern. SimM can run only on PC-ATs or compatibles under MS-DOS currently. However, being written in C without using non-portable functions, SimM is portable to any machine with a C compiler.

The designers of SimM are working on revisions to SimM. One of the goals is to eliminate the memory limitation in the current version of SimM.

Appendix D PCU Microinstruction Mnemonic and CMDASM Listing

ACS		PE Accummulator select
AWE		PE Accumulator write enable
PCLK		Generate processor clock
PCS		Parameter memory select
PWE		Parameter memory write enable
PRMACC		Accumulate parameter
LDPRM		Load parameter accumulator
INCPRM		Increment parameter accumulator
PAD	address	Address parameter memory
OP	opcode	Generate PE board opcode
PHASE		Set PE board operation phase
INCWA		Increment weight address
INCAA		Increment PE accumulator address
LDWA	address	Load weight address
LDAA	address	Load PE accumulator address
AOE		Enable PE accumulator memory output
IMEDDIS		Disable instruction immediate field
IMD	data	Generate immediate data
INV	data	Generate inverted immediate data
RDWM		Read weight memory
WRWM		Write weight memory
SUM2P		Select parameter accumulator for parameter input
CTR2P		Select loop counter for parameter input
EXT2P		Select external input for parameter input
OUTPRQ	prq	Output parameter request
AASEL		Select weight address for accumulator address
NOP		Continue
JMP	address	Unconditional jump
DJZ	address	Decrement loop counter & jump on zero
JPACK	address	Jump if parameter acknowledge
JPCY	address	Jump if parameter accumulator carry
DECCTR		Decrement loop counter
LDCTR	value	Load loop counter
SRRD		Shift register memory read
SRWR		Shift register memory write
SRLTCH		Latch shift register memory data
SRXD		Latch exchange data into shift register
RDPRM	address	Read parameter memory
WRPRM	address	Write parameter memory

Table C.1 Mnemonic PCU Instruction Set

```
OPFIELD opcodes[] = {
/*
OPCODE      MASK FIELD                CODE FIELD                TYPE */
{"ACS",     {0x00,0x00,0x00,0x80,0x04}, {0x00,0x00,0x00,0x00,0x04}, NO_OP },
{"AWE",     {0x00,0x00,0x00,0x80,0x20}, {0x00,0x00,0x00,0x00,0x20}, NO_OP },
{"PCLK",    {0x00,0x00,0x00,0x80,0x10}, {0x00,0x00,0x00,0x00,0x10}, NO_OP },
{"PCS",     {0x00,0x00,0x00,0x80,0x05}, {0x00,0x00,0x00,0x80,0x05}, NO_OP },
{"PWE",     {0x00,0x00,0x00,0x80,0x20}, {0x00,0x00,0x00,0x80,0x20}, NO_OP },
{"PRMACC",  {0X00,0x00,0x00,0x80,0x18}, {0X00,0x00,0x00,0x80,0x10}, NO_OP },
{"LDPRM",   {0X00,0x00,0x00,0x80,0x18}, {0X00,0x00,0x00,0x80,0x18}, NO_OP },
{"INCPRM",  {0X00,0x00,0x00,0x80,0x18}, {0X00,0x00,0x00,0x80,0x08}, NO_OP },
{"PAD",     {0x00,0x00,0x00,0x8F,0x00}, {0x00,0x00,0x00,0x80,0x00}, LOWOP },
{"OP",      {0x00,0x00,0x00,0x8F,0x00}, {0x00,0x00,0x00,0x00,0x00}, LOWOP },
{"PHASE",   {0x00,0x00,0x00,0x80,0x08}, {0x00,0x00,0x00,0x00,0x08}, NO_OP },
{"INCWA",   {0x00,0x00,0x00,0x80,0x40}, {0x00,0x00,0x00,0x00,0x40}, NO_OP },
{"INCAA",   {0x00,0x00,0x01,0x80,0x00}, {0x00,0x00,0x01,0x00,0x00}, NO_OP },
{"LDWA",    {0x7F,0xFF,0x00,0x80,0x40}, {0x00,0x00,0x00,0x80,0x40}, HI_15 },
{"LDAA",    {0x07,0xFF,0x01,0x80,0x00}, {0x00,0x00,0x01,0x80,0x00}, HI_11 },
{"AOE",     {0x00,0x00,0x00,0x00,0x80}, {0x00,0x00,0x00,0x00,0x80}, NO_OP },
{"IMEDDIS", {0x00,0x00,0x00,0x00,0x01}, {0x00,0x00,0x00,0x00,0x01}, NO_OP },
{"IMD",     {0xFF,0xFF,0x00,0x00,0x00}, {0x00,0x00,0x00,0x00,0x00}, HI_OP },
{"INV",     {0xFF,0xFF,0x00,0x00,0x00}, {0x00,0x00,0x00,0x00,0x00}, INVOP },
{"RDWM",    {0x00,0x00,0x06,0x80,0x00}, {0x00,0x00,0x02,0x00,0x00}, NO_OP },
{"WRWM",    {0x00,0x00,0x06,0x80,0x00}, {0x00,0x00,0x04,0x00,0x00}, NO_OP },
{"SUM2P",   {0x00,0x00,0x06,0x80,0x00}, {0x00,0x00,0x00,0x80,0x00}, NO_OP },
{"CTR2P",   {0x00,0x00,0x06,0x80,0x00}, {0x00,0x00,0x02,0x80,0x00}, NO_OP },
{"EXT2P",   {0x00,0x00,0x04,0x80,0x00}, {0x00,0x00,0x04,0x80,0x00}, NO_OP },
{"OUTPRQ",  {0x80,0x00,0x06,0x80,0x00}, {0x00,0x00,0x06,0x80,0x00}, HIBIT },
{"AASEL",   {0x00,0x00,0x00,0x00,0x02}, {0x00,0x00,0x00,0x00,0x02}, NO_OP },
{"NOP",     {0x00,0x00,0x00,0x00,0x00}, {0x00,0x00,0x00,0x00,0x00}, NO_OP },
{"JMP",     {0x07,0xFF,0xE0,0x00,0x00}, {0x00,0x00,0x20,0x00,0x00}, HI_11 },
{"DJZ",     {0x07,0xFF,0xE0,0x00,0x00}, {0x00,0x00,0x40,0x00,0x00}, HI_11 },
{"JPACK",   {0x07,0xFF,0xE0,0x00,0x00}, {0x00,0x00,0x60,0x00,0x00}, HI_11 },
{"JPCY",    {0x07,0xFF,0xE0,0x00,0x00}, {0x00,0x00,0x80,0x00,0x00}, HI_11 },
{"DECCTR",  {0x00,0x00,0xE0,0x00,0x00}, {0x00,0x00,0xC0,0x00,0x00}, NO_OP },
{"LDCTR",   {0x07,0xFF,0xE0,0x00,0x00}, {0x00,0x00,0xE0,0x00,0x00}, INV9_14},
{"SRRD",    {0x00,0x00,0x00,0x70,0x00}, {0x00,0x00,0x00,0x10,0x00}, NO_OP },
{"SRWR",    {0x00,0x00,0x00,0x70,0x00}, {0x00,0x00,0x00,0x30,0x00}, NO_OP },
{"SRLTCH",  {0x00,0x00,0x00,0x70,0x00}, {0x00,0x00,0x00,0x50,0x00}, NO_OP },
{"SRXD",    {0x00,0x00,0x00,0x70,0x00}, {0x00,0x00,0x00,0x70,0x00}, NO_OP },
{"RDPRM",   {0x00,0x00,0x00,0x8F,0x05}, {0x00,0x00,0x00,0x80,0x05}, LOWOP },
{"WRPRM",   {0x00,0x00,0x00,0x8F,0x25}, {0x00,0x00,0x00,0x80,0x25}, LOWOP }
};
```

Table C.2 CMDASM Instruction Difinition File

```
/* CMDASM (PCU microinstruction asmebler) */
#include <stdio.h>
#include <dos.h>
#include <string.h>
#include <ctype.h>
#include <stdarg.h>
#include <stdlib.h>
#include "cmdasm.h"
#include "opcodes.h"
#define VERSION "3.3, 07/31/91"
#definelastchar(s) ((s)[strlen(s)-1])
#defin istknchr(c)(isalnum(c) || ((c)=='_'))
#define ALERT(str,arg)    { fprintf(stderr,str,arg); fcloseall(); exit(-1); }
#define ARRAYSIZE(array)  (sizeof(array)/sizeof(array[0]))
#define NONE    0
#defin  FALSE   0
#defin  TRUE    1
#define GETWORD  100
#defin  GETLINE  101
#define EOFILE  -1
#define EOLINE   0
```

```
FILE *fi, *fol, *foo;
int     firstobjrec = TRUE;
long getnum();
void putop(ins_word, long, int ,int);
void zero_ins(ins_word);
void ins_cpy(ins_word, ins_word);
char* ins2str(ins_word);
main (argc, argv)
int argc;
char *argv[];
{
   char   line[101], word[101];
   char   symlab[101], numstr[101], *dummy;
   char   *error;
   int    i, lc, lincnt, status, adding;
   int    gotsym, symnum, symcnt=0;
   int    foundop, gotcode=FALSE;
   ins_word insmsk, inswrd, tmpwrd;
   long   operand;
   SYMREC slrec[1000];
#ifdef DEBUG
        testops();
#endif
        zero_ins(insmsk);
        zero_ins(inswrd);
   --argc;
   *(strchr(argv[0],'.'))  =  NULL;
   argv[0]  =  strrchr(argv[0],'\\') + 1;
   fprintf (stderr, " %s  --  Version %s\n", strupr(argv[0]), VERSION);
   if ( argc<1 || argc>3 )  {
      fprintf (stderr, "    Needs one to three arguments:  the source ");
      fprintf (stderr, "input file, the list output\n    file, and the ");
      fprintf (stderr, "object output file.  A default extension of \".");
      fprintf (stderr, "CMD\" will be\n    assummed if none is provided ");
      fprintf (stderr, "for the input file.  If no output filename is\n");
      fprintf (stderr, "    given the default filename is the same as ");
      fprintf (stderr, "the input file.  The default\n    extensions are");
      fprintf (stderr, " \".LST\" and \".OBJ\".\n");
      exit (-1);
   }
   openfiles (argc, argv);

   lincnt  =  0;
   lc  =  0;
   fprintf (stderr, " Pass 1:\n");
        while( (status = getstr(GETLINE,line,100,fi))  !=  EOFILE )  {
      ++lincnt;
      while ( status == EOLINE )  {
         status  =  getstr (GETLINE, line, 100, fi);
         ++lincnt;
      }
      gotsym  =  FALSE;
      if ( isalpha(line[0]) )  {
         getstr (GETWORD, symlab, 100, fi);
            symlab[16]  =  '\0';  /*if string more than 16 chars, shorten*/
         for (symnum=0; (symnum<symcnt) && !gotsym; ++symnum) {
            if ( !strcmpi(symlab,slrec[symnum].sym) )  {
            fprintf (stderr, "  WARNING:  Redefinition of symbol/label '%.16s' in line %d.\n", symlab, lincnt);
            gotsym  =  TRUE;
            }
         }
         strcpy (slrec[symnum].sym, symlab);
            if ( symnum == symcnt )  /*symbol not found in table*/
               ++symcnt;
         gotsym  =  TRUE;
      }
         if ( line[0] != '+' )  {
         if ( getstr(GETWORD,word,100,fi) != EOLINE )  {
         if ( word[0] == '.' )  {
         if ( getstr(GETWORD,numstr,100,fi) != EOLINE )  {
         operand  =  strtol(numstr, &dummy, 0);
         if ( !errno )  {
         if ( !strcmpi(word,".ORG"))  lc  =  operand;
         els   if ( !strcmpi(word,".EQU") )  {
         if (gotsym)  slrec[symnum].val = operand;
         else {
                        fprintf (std  rr, "  ERROR:  No symbol in line %d.\n", lincnt);
         if ( symnum == symcnt )  --symcnt;
                     }
                  }
```

```
            }
        else if ( gotsym ) {
                fprintf (stderr, "  ERROR: Could not decode value in line %d.\n", lincnt);
        if ( symnum == symcnt ) --symcnt;
            }
          }
        else if ( gotsym ) {
                fprintf(stderr, "  ERROR: Expected value but none found in line %d.\n", lincnt);
        if ( symnum == symcnt ) --symcnt;
            }
          }
        else
        for (i=0; i<ARRAYSIZE(opcodes); ++i)
        if ( !strcmpi(word,opcodes[i].mnem) ) {
                if ( gotsym )
                  slrec[symnum].val = (long)lc;
                ++lc;
            }
        }
        else if ( gotsym )
        {
          fprintf (stderr, "  ERROR:  No opcode/command in line %d.\n", lincnt);
        if ( symnum == symcnt )
          --symcnt;
        }
      }
    }

/* pass two of assembly*/
    rewind (fi);
        lincnt = 0;
    lc = 0;
    fprintf (stderr, " Pass 2:\n");
    while ( (status = getstr(GETLINE,line,100,fi)) != EOFILE ) {
      ++lincnt;
      if ( isalpha(line[0]) )
          getstr (GETWORD, word, 100, fi);
      if ( status == EOLINE )
          fprintf (fol, "                    : %s\n", line);
      else {
          adding = (line[0]=='+');
          foundop = FALSE;
          error = NONE;
          if ( getstr(GETWORD,word,100,fi) != EOLINE ) {
            if ( word[0] == '.' ) {
            if (  !strcmpi(word,".ORG")  &&  getstr(GETWORD,numstr,100,fi) != EOLINE )
            {
                  operand = strtol(numstr, &dummy, 0);
            if ( !errno )
            {
                    if ( gotcode )
                      addtoobj (lc, inswrd);
                    gotcode = FALSE;
            lc = operand;
            zero_ins(insmsk);
            zero_ins(inswrd);
                  }
              }
            } else {
              for (i=0; (i<ARRAYSIZE(opcodes) && !foundop); ++i)
                if ( !strcmpi(word,opcodes[i].mnem) ) {
                  foundop = TRUE;
            ins_cpy(tmpwrd,opcodes[i].code);
            if ( opcodes[i].type != NO_OP )
            {
            if ( getstr(GETWORD,numstr,100,fi) != EOLINE )
            {
            operand = getnum(numstr, slrec, symcnt, &error);
                          if ( opcodes[i].type == LOWOP )
            putop(tmpwrd, operand, 8 , 4);
                          else if (opcodes[i].type == HI_OP )
            putop(tmpwrd, operand, 24 ,16);
                          else if ( opcodes[i].type == INVOP )
            putop(tmpwrd , ~operand, 24, 16);
            els  if ( opcodes[i].type == HIBIT)
            putop(tmpwrd , op  rand, 39 , 1);
            else if (opcodes[i].type == HI_15)
            putop(tmpwrd , operand , 24 , 15);
            else if (opcodes[i].type == HI_11)
            putop(tmpwrd , operand , 24 , 11);
```

```
        else if (opcodes[i].type == INV9_14)
        putop(tmpwrd , ~operand , 33 , 6);
                    }
                    else
                       error = "Expected operand and found none";
                 }
        if ( !error )
        {
        if ( adding )
        {
        if ( add_ins(inswrd , tmpwrd , insmsk , opcodes[i].mask) != 0)
        error = "Bitmask conflict";
        else
        gotcode = TRUE;
                    }
                    else {
                       if ( gotcode ) {
                          addtoobj (lc, inswrd);
                          ++lc;
                       }
                    gotcode = TRUE;
         ins_cpy(insmsk , opcodes[i].mask);
         ins_cpy(inswrd , tmpwrd);
                    }
                 }
              }
           if ( !foundop )
              error = "Unknown opcode";
         }
        fprintf (fol, "%03X %s:%s\n", lc, ins2str(inswrd),line);
          if ( error ) {
             fprintf (fol, "  ERROR:  %s.\n\n", error);
             fprintf (stderr, "   ERROR:  %s in line %d.\n", error, lincnt);
          }
        }
      }
   }
   addtoobj (lc, inswrd);

        fprintf (fol, "\n\n\n");
        for (symnum=0; symnum<symcnt; ++symnum ) {
        if (lastchar(slrec[symnum].sym) == '_') {
        fprintf (fol,"#define %-16s 0x%lX\n", slrec[symnum].sym, slrec[symnum].val);
        }
        }
   fcloseall();
   exit (0);
}



openfiles (count, names)
int  count;
char *names[];
{
   int  dotpos;
   char *dotloc, source[101], list[101], object[101];

   strcpy (source, strupr(names[1]));
   if ( (dotloc = strchr(source,'.')) != (char *)NULL )
      dotpos = dotloc - source;
   else {
      dotpos = strlen(source);
      strcat (source, ".CMD");
   }
   if (--count)
      strcpy (list, strupr(names[2]));
   else {
      strncpy (list, source, dotpos);
      list[dotpos] = NULL;
   }
   if ( (dotloc = strchr(list,'.')) != (char *)NULL )
      dotpos = dotloc - list;
   else {
      dotpos = strlen(list);
      strcat (list, ".LST");
   }
```

```
  if (count == 2)
     strcpy (object, strupr(names[3]));
  else {
     strncpy (object, list, dotpos);
     object[dotpos] = NULL;
  }
  if ( !strchr(object,'.') )
     strcat (object, ".OBJ");


  if ( (fi = fopen(source, "r")) == (FILE *)NULL ) {
     fprintf (stderr, "FATAL ERROR:  Could not open %s.\n", source);
     fcloseall();
     exit(-1);
  }
  if ( (fol = fopen(list, "w")) == (FILE *)NULL ) {
     fprintf (stderr, "FATAL ERROR:  Could not open %s.\n", list);
     fcloseall();
     exit(-1);
  }
  if ( (foo = fopen(object, "w")) == (FILE *)NULL ) {
     fprintf (stderr, "FATAL ERROR:  Could not open %s.\n", object);
     fcloseall();
     exit(-1);
         } else {
         firstobjrec = TRUE;
         }
}


getstr (action, string, length)
int  action;
char *string;
int  length;
{
   static char line[501];
   static int  l, w, len;
   switch (action) {
      case GETLINE:
         if ( fgets(line,length,fi) )
            len  =  strlen (line);
         else
            return (EOFILE);
         for (l=0; l<len; ++l)
            if ( line[l] == '\n' ) {
               line[l]  =  '\0';
               break;
            }
         strcpy (string, line);
         for (l=0; l<len; ++l)
            if ( line[l] == ';' )
            return (EOLINE);
          else if ( line[l]=='.' || istknchr(line[l]) )
               return (len);
         return (EOLINE);
       case GETWORD:
         if ( line[l] == ';' )
            return (EOLINE);
         w  =  0;
         while ( l<len && w<length ) {
          if ( line[l]=='.' || istknchr(line[l]) )
               string[w++]  =  line[l];
            else if ( line[l]=='+' || line[l]==' ' || line[l]=='\t' || line[l]==';' )
               break;
            ++l;
         }
         string[w]  =  '\0';
         while ( l < len ) {
          if ( line[l]=='.' || istknchr(line[l]) || line[l]==';' )
               break;
            ++l;
         }
         return (w);
       default:
         break;
   }
}
```

```
addtoobj (loccnt, insword)
int   loccnt;
ins_word  insword;
{
int i;
        if (firstobjrec == TRUE) firstobjrec = FALSE;
        fprintf (foo, " db 0%02XH,0%02XH", loccnt & 255, (loccnt >> 8) & 255);
        for( i=NUMCHARS-1; i >= 0; i--) {
          fprintf (foo ,",0%02XH", insword[i]);
        }
        fprintf(foo,"\n");
}

long getnum (string, record, count, errstr)
char   *string;
SYMREC  record[];
int    count;
char    **errstr;
{
   int  i;
   long  operand;
   char  *dummy;
        if (isalpha(string[0]))  {
     string[16]  =  '\0';
        for (i=0; i<count; ++i) {
        if ( !strcmpi(string,record[i].sym)) return (record[i].val);
        }
        *errstr  =  "No match for symbol/label (0 used instead)";
     return (0L);
        } else {
     operand  =  strtol(string, &dummy, 0);
     if ( errno )  {
        *errstr  =  "Unable to decode operand (0 used instead)";
        operand  =  0L;
     }
        return (operand);
   }
}


int add_ins( dest_wrd , src_wrd , dest_msk , src_mask)
/* routine checks for bit conflicts in and if there are none
then or's src arguments into dest arguments.*/

ins_word dest_wrd, src_wrd , dest_msk , src_mask;
{
  int i=0, bad=0;
  while ( (i < NUMCHARS) && !bad) {
        bad = ((dest_wrd[i] ^ src_wrd[i]) & dest_msk[i] & src_mask[i]);
    i++;
  }
  if (!bad)
        for (i=0; i<NUMCHARS ; i++)
        {
        dest_wrd[i] |= src_wrd[i];
        dest_msk[i] |= src_mask[i];
        }
  return (bad);
}



void putop( dest_ins , operand , startbit , numbits)
ins_word dest_ins;
long operand;
int startbit , numbits;
/* routine accepts an operand and the starting bit (least significant bit
index) and number of bits that it occupies in the instruction word.
It puts each operand bit in th  instruction word one at a tim  (not
very elegant.) */
{
int bitpos, charnum , tmpchar , i;

  i = 0;     /* bit position within operand*/
  bitpos = startbit;  /*bit position within ins_word*/
  do {
        charnum = (WORDLENGTH-bitpos - 1) / 8;  /*array index into ins_w  rd*/
        tmpchar = 1 << ((bitpos) % 8);
    if (operand & ( 1 << i) )
        dest_ins[charnum] |= tmpchar;
```

```
    else
        dest_ins[charnum] &= ~tmpchar;
  } while (++bitpos, ++i < numbits) ;
}

void zero_ins( word )
/* Sets every bit in the instruction word to a zero. Object oriented
programming is our friend.*/
ins_word word;
{
  int i;
  for (i=0; i < NUMCHARS; i++)
        word[i] = 0;
}

void ins_cpy (dest_wrd , src_wrd)
/* dest_wrd = src_wrd */
ins_word dest_wrd , src_wrd;
{
  int i;
  for (i=0; i<NUMCHARS; i++)
        dest_wrd[i] = src_wrd[i];
}
char* ins2str( word )
/* Converts instruction word into an ascii, hex string.*/
ins_word word;
{
  static char digits[NUMCHARS * 2 + 1] ;

  int i;
  for(i=0; i<NUMCHARS ; i++)
        sprintf(digits + i*2 ,"%02X", word[i]);
  return digits;
}

#ifdef DEBUG
testops()
/*prints out the opcodes array to see if it was initialized correctly.*/
{
  int i,j;
  for (j=0; j< 15; j++)
  {
        printf("%s\n", opcodes[j].mnem);
        for (i=0; i<NUMCHARS ; i++)
        printf("%d- %X  , %X\n", i,opcodes[j].mask[i] , opcodes[j].code[i]);
  }
}
#endif
```

Appendix E Prototype PCU Macroinstructions in C Language

```
      ;============== MOVE
PUTA(aa, sa)          AM <== SM
GETA(sa, aa)          SM <== AM
PUTW(wa, sa)          WM <== SM
GETW(sa, wa)          SM <== WM
MOVA(aa, aa2)         AM <== AM(2)
MOVP(sa, sa2)         SM <== SM(2)
A2W(wa, aa)           WM <== AM
W2A(aa, wa)           AM <== WM
SFT(sa, w)            Circular SM, for w PEs
      ;============== Arithematic
ADDW(aa, wa)          AM <== AM + WM
ADDA(aa, aa2)         AM <== AM + AM(2)
INCA(aa)              AM <== AM + 1
ANDW(aa, wa)          AM <== AM & WM
ANDA(aa, aa2)         AM <== AM & AM(2)
SUBW(aa, wa)          AM <== AM - WM
SUBA(aa, aa2)         AM <== AM - AM(2)
DECA(aa)              AM <== AM - 1
XORW(aa, wa)          AM <== AM ^ WM
XORA(aa, aa2)         AM <== AM ^ AM(2)
NOTA(aa)              AM <== ~AM
XMAC(aa, wa, off)     AM <== Sum(AM + P * WM) (inner product)
MACX(aa, wa, off, sa) AM,SM <== Sum(sm + P * WM)
XPRDCT(wa, wa2, off, scale) outter product
MAC(aa, wa)           AM <== AM + P * WM
MUL(aa, wa)           AM <== P * WM
ASL(aa, b)            AM <== AM << b
OADDA(aa, aa2)        AM <== AM + AM(2) with overflow checking
OSUBA(aa, aa2)        AM <== AM - AM(2) with overflow checking
OSUBW(aa, wa)         AM <== AM - Wm  with overflow checking
OADDW(aa, wa)         AM <== AM + Wm  with overflow checking
SGEXT(aa, nbits)      Sign extend (Np bits to nbits bits)
ADDWW(wa, wa2, nwbit, oldbits)  WM <== WM(nwbits) + WM2(oldbits)
      ;============== Conditional
CLA(aa)               Clear AM
CLW(wa)               Clear WM
CLAZ(aa)              Clear AM if Zero
CLAC(aa)              Clear AM if Carry
CLAS(aa)              Clear AM if Sign
CLAN(aa)              Clear AM if Negative
CLANZ(aa)             Clear AM if Not Zero
CLANC(aa)             Clear AM if Not Carry
CLANS(aa)             Clear AM if Not Sign
CLAP(aa)              Clear AM if Positive
W2A(aa, wa)           WM to AM
W2AZ(aa, wa)          WM to AM if Zero
W2AC(aa, wa)          WM to AM if Carry
W2AS(aa, wa)          WM to AM if Sign
W2AN(aa, wa)          WM to AM if Negative
W2ANZ(aa, wa)         WM to AM if Not Zero
W2ANC(aa, wa)         WM to AM if Not Carry
W2ANS(aa, wa)         WM to AM if Not Sign
W2AP(aa, wa)          WM to AM if Positive
      ;============== Set Parameter
SETC                  Set Carry
CLRC                  Clear Carry
SETZ                  Set Zero
SETK(x)               Set # of PE, K = x
SETLEN(x)             Set precision of result of mul., len =x
SETNW(x)              Set precision of WM, Nw = x
SETNP(x)              Set precision, Np = x
      ;============== MISC
LIMX(aa, nbits, oldbits) limit AM from oldbits to nbits (oldbits>=nbits)
ROUND(aa)             round @aa (for Np bits)
```

Appendix F Sample PCU microinstruction Subroutine

```
tempWA: .EQU   0
X:     .EQU   0       ;just a dummy
PE:    .EQU   0x1000  ;DEFAULT PEs/pRING 8 0xefff
Sboard: .EQU   0      ;SA=AA
pbits:  .EQU   0x2000  ;Np            16 0xefff
wbits:  .EQU   0x1e00  ;Nw-1          15 0xf1ff
tlen:   .EQU   0x4000  ;Np+Nw         32 0xdfff
GUARD:  .EQU   6       ;guard bits = 6
;------Define Parameter addresses ---
parm5:  .EQU   8
parm6:  .EQU   9
len:    .EQU   10
Nw:     .EQU   11
Npsave: .EQU   12
ksave:  .EQU   13
tempAA: .EQU   14
retx:   .EQU   15

parm4:  .EQU   7
parm3:  .EQU   3
parm2:  .EQU   2
parm1:  .EQU   1
istart: .EQU   0
offset: .EQU   3       ;parm3
WA:     .EQU   2       ;parm2
tarAA:  .EQU   1       ;parm1
k:      .EQU   4
Np:     .EQU   5
SA:     .EQU   6
;============== ADDWW @w1,@w2 ========================
addww:  rdprm   parm1   ;move first addend from weight
        ldwa    X       ; to accum tmp location
+       rdprm   tempAA  ; (tmp-1 actually)
        ldaa    X
+       rdprm   Npsave  ;load precision
        ldctr   X       ; to counter
        jmp     adww2   ;middle of move W -> Atmp loop
+       rdwm            ;prefetch weight memory
adww1:  op      0       ;AD <- Y
+       acs             ;write data to accum
+       awe
+       djz     adww3   ;move complete -> exit lp
+       incwa           ;increment W address
+       rdwm            ;prefetch nxt wt bit
adww2:  rdwm            ;fetch bit from wt mem
+       op      4       ;Y <- WI
+       phase
+       pclk
+       incaa           ;increment AA address
+       jmp     addww1  ;jmp to write phase
adww3:  rdprm   parm2   ;get adr of other addend
        ldwa    X       ; in wt mem
+       rdprm   tempAA  ;dest & other addend now in Atmp
        ldaa    X
+       rdprm   Npsave
        ldctr   X       ;load # of bits
+       op      0       ;init carry
+       pclk
        rdwm            ;prefetch weight data
+       jmp     adww5   ;middle of add loop
adww4:  op      10      ;AD <- sum (standard add)
+       pclk            ; & update flags
+       acs             ;write result to accum .
+       awe
+       djz     adww6   ;when add done exit ti move result
+       incwa           ;increment to next wt
+       rdwm            ;prefetch wt value
adww5:  incaa           ;pre-inc accum addr (inc then load)
+       rdwm            ;fetch addends (w  ight)
+       acs             ; (and accumulator)
+       aoe
+       op      9       ;WO <- WI, Y <- AD (load)
+       phase           ;load phase
+       pclk            ;latch WO,Y
+       jmp     adww4   ;g  to operate phase
adww6:  rdprm   parm1   ;move result to 1st opm
        ldwa    X       ; from accum tmp location
+       rdprm   tempAA  ; (tmp-1 actually)
```

```
      ldaa   X
+     rdprm  Npsave  ;load precision
      ldctr  X       ; to counter
adww7: incaa         ;pre-inc Atmp address
+     acs            ;read Amem
+     aoe
+     op     0       ;Y <- AD
+     phase          ; (load phase)
+     pclk
      op     6       ;WO <- Y
+     wrwm           ;write it wt mem
+     djz    setq    ;cleanup after move (sign->Q)
      incwa          ;else inc to nxt wt mem addr
+     jmp    adww7   ; loop for nxt bit

setq:  nop
```

WHAT IS CLAIMED IS:

1. A system for processing digital data, comprising:

a plurality to processing elements arranged in at least two interconnected processor arrays disposed about a communications network:

means for delivering data to said plurality of processing elements: and

means for implementing processing of said data synchronously and parallely by said plurality for processing elements within each of said interconnected processor arrays.

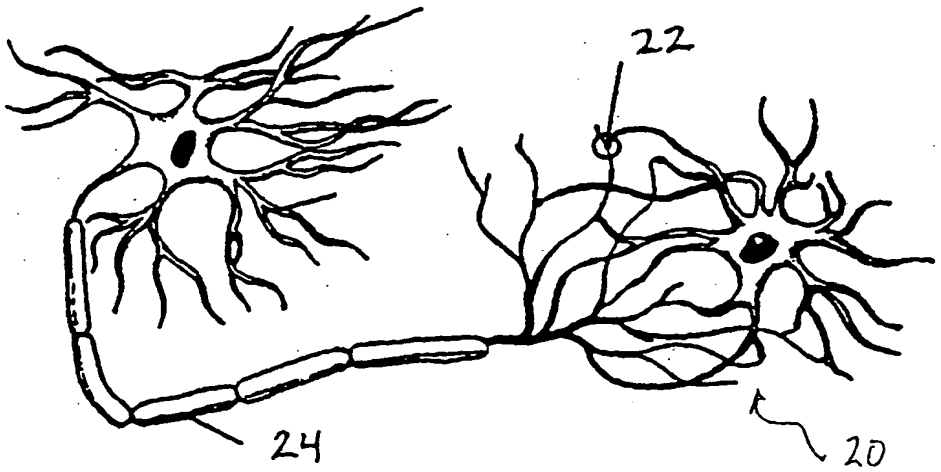


Figure 1

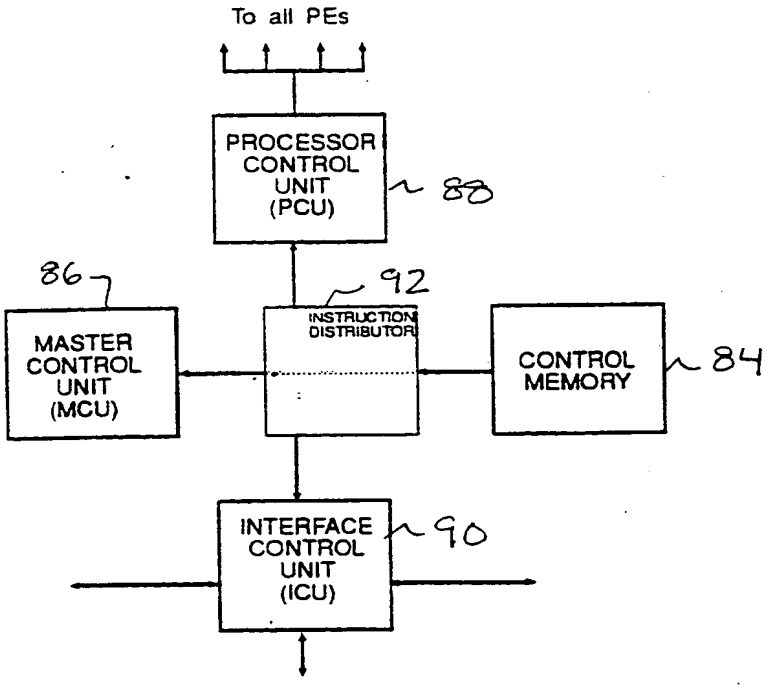


Figure 6

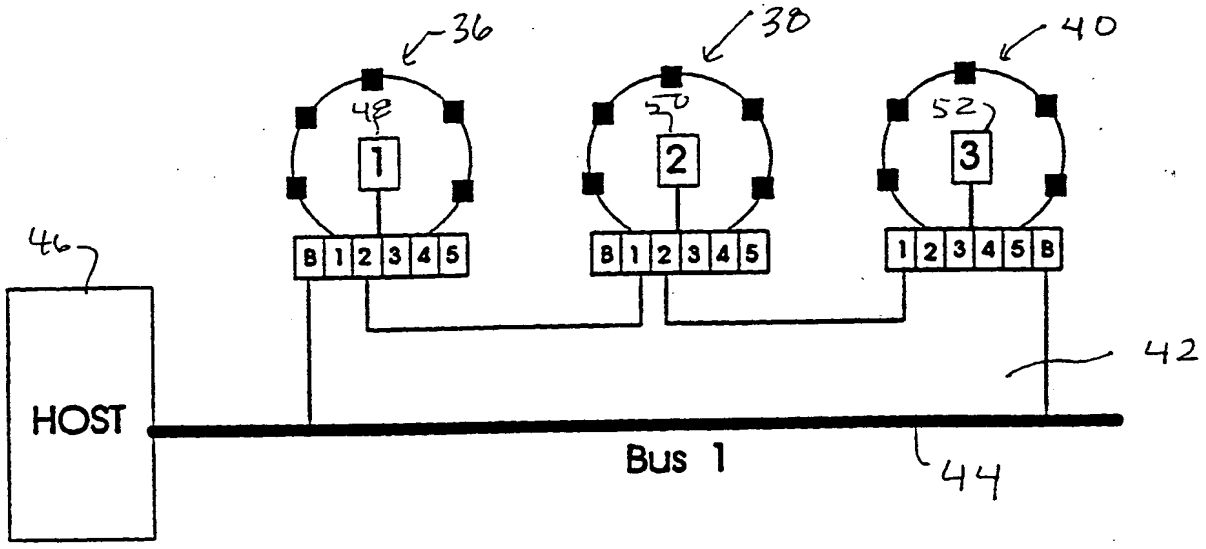


Figure 3.

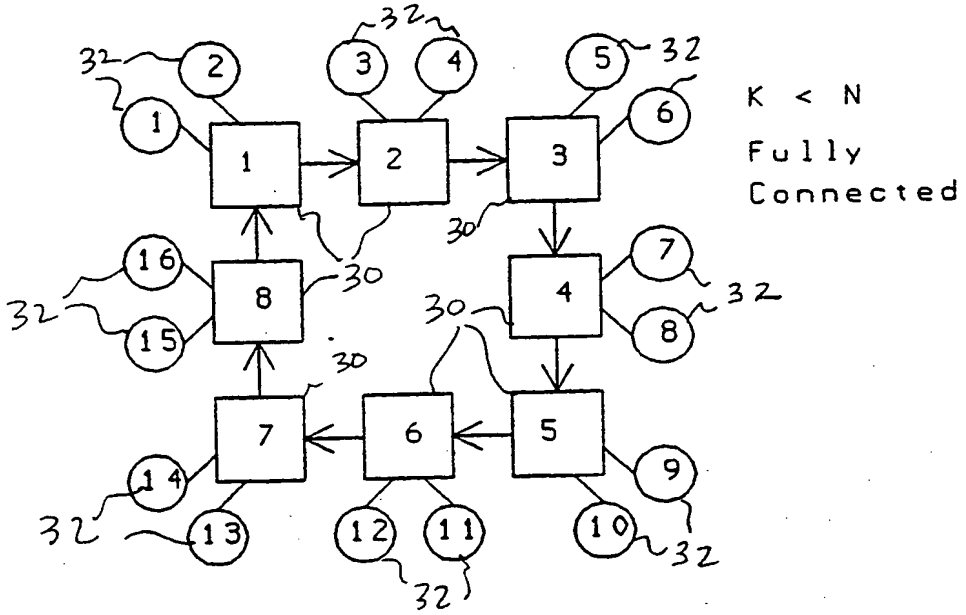


Figure 2.

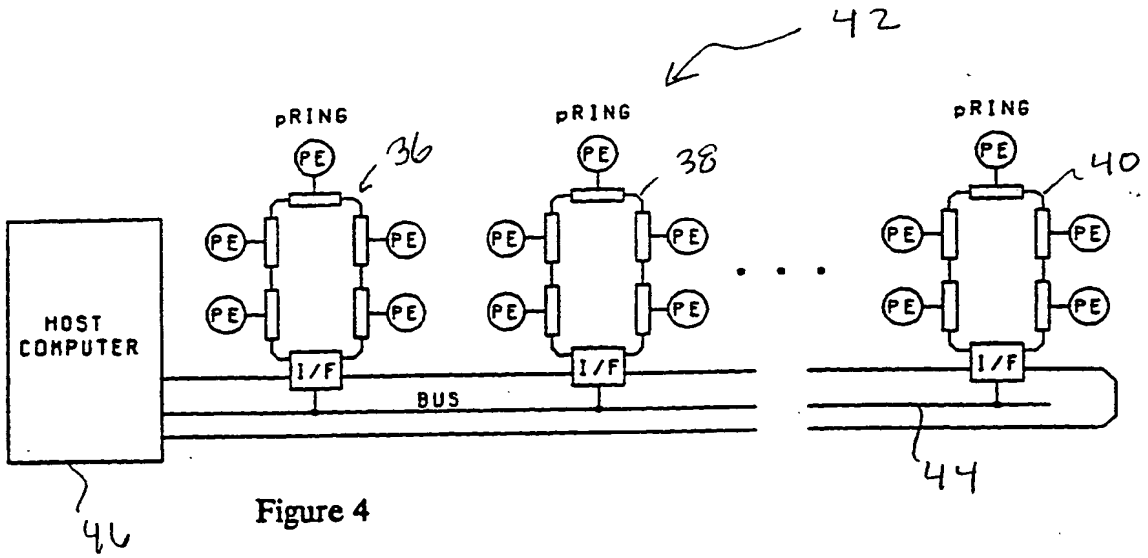


Figure 4

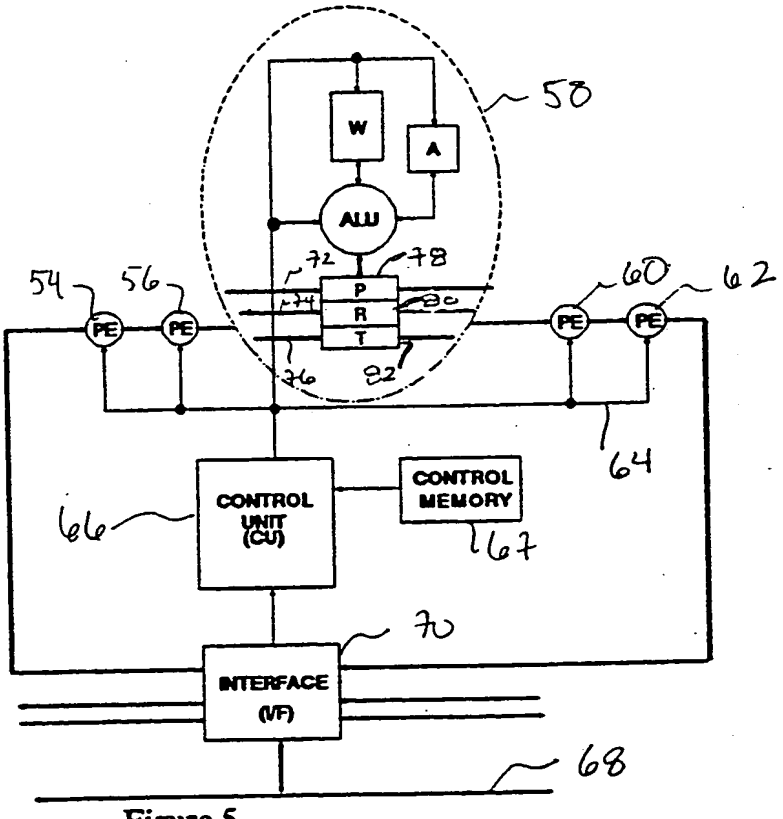


Figure 5

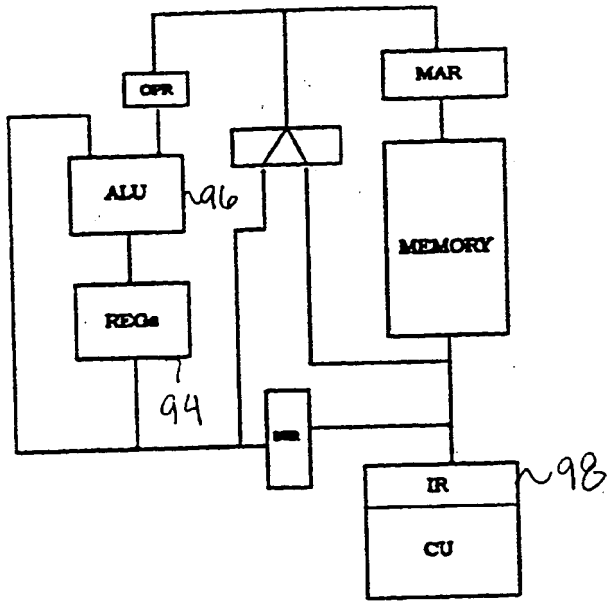


Figure 7

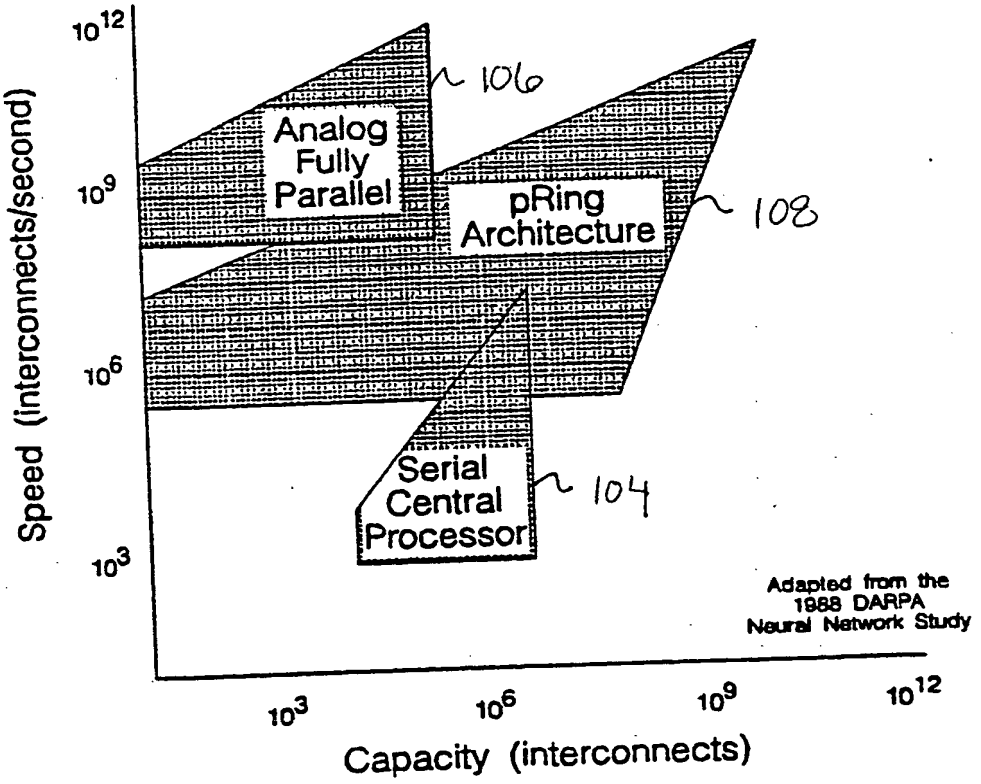


Figure 9

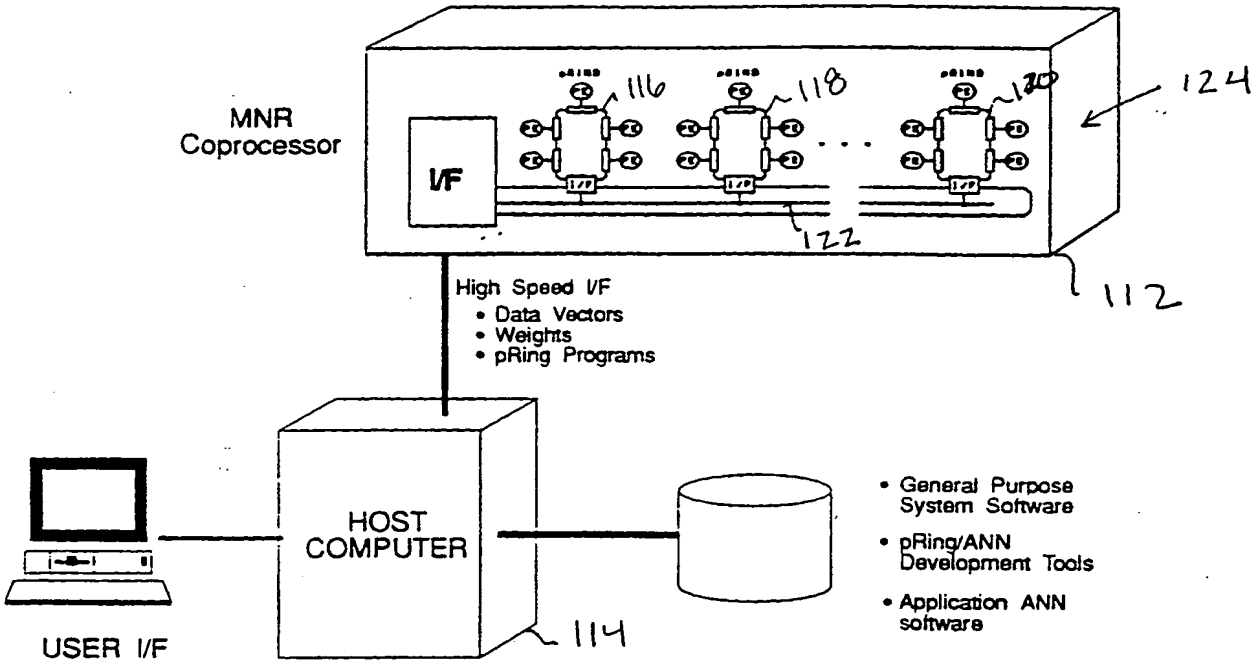


Figure 10

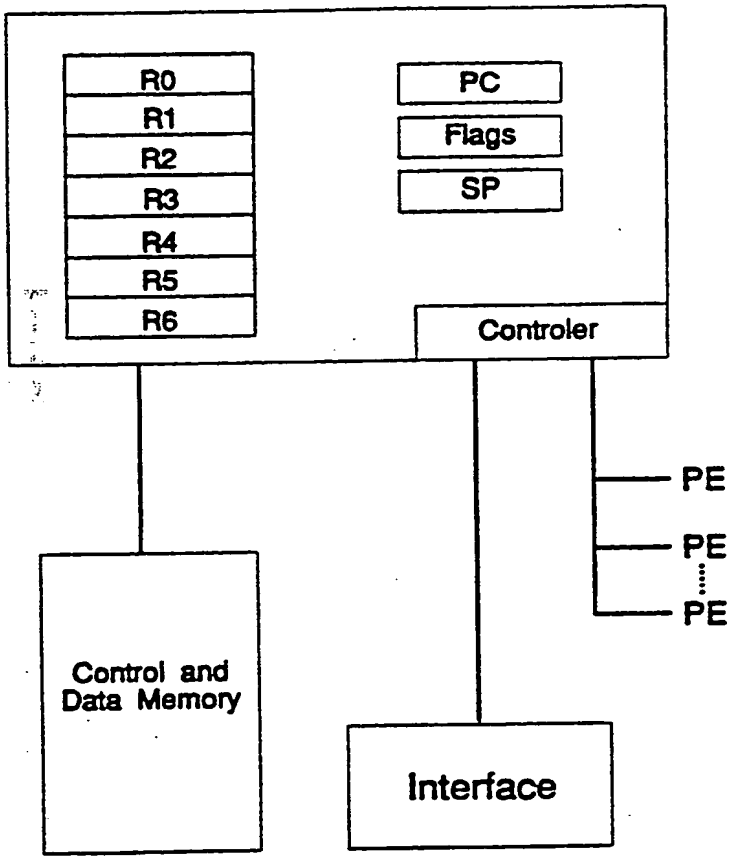
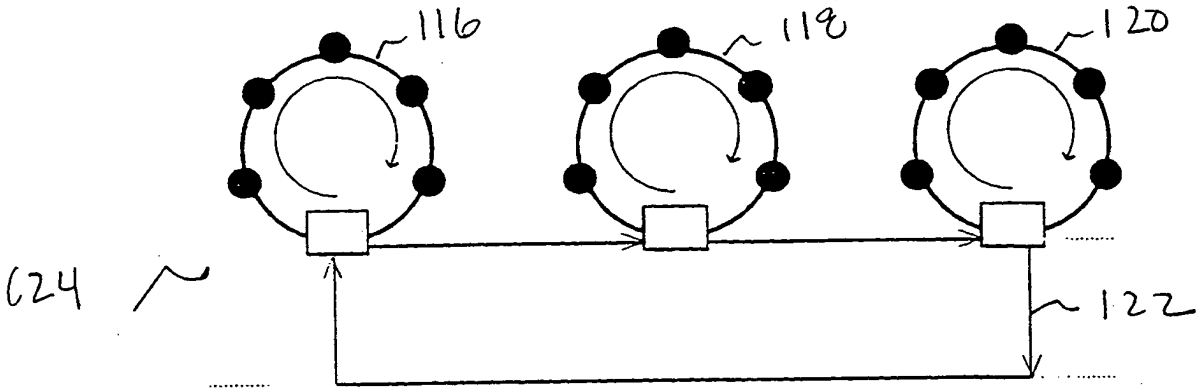


Figure 8

FIG. 11



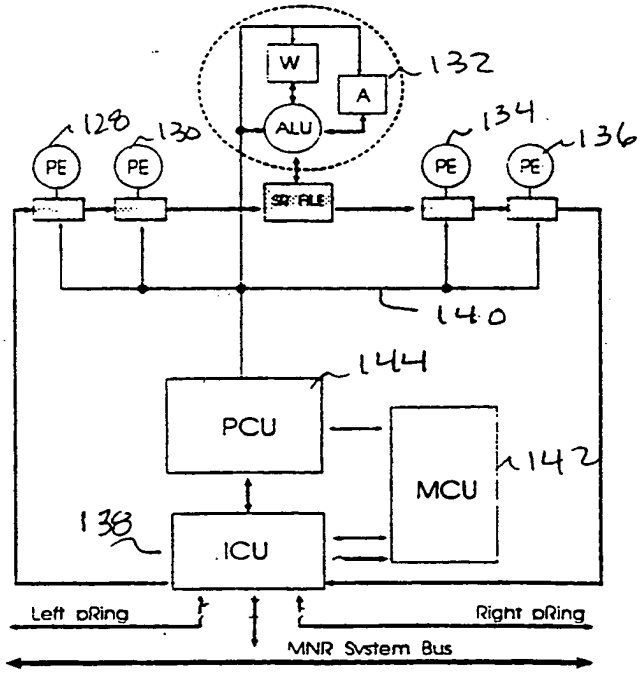


FIG. 12

FIG. 13

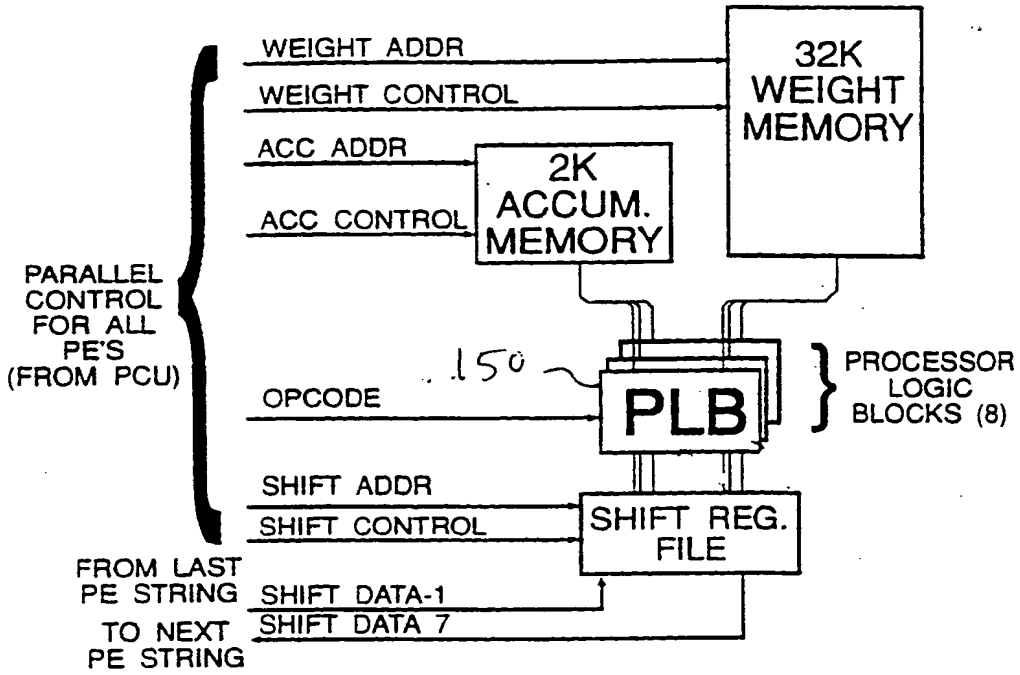

WEIGHT ADDR
WEIGHT CONTROL
32K WEIGHT MEMORY
ACC ADDR
2K ACCUM. MEMORY
ACC CONTROL
PARALLEL CONTROL FOR ALL PE'S (FROM PCU)
150
OPCODE
PLB
PROCESSOR LOGIC BLOCKS (8)
SHIFT ADDR
SHIFT CONTROL
SHIFT REG. FILE
FROM LAST PE STRING
SHIFT DATA-1
TO NEXT PE STRING
SHIFT DATA 7

FIG. 14

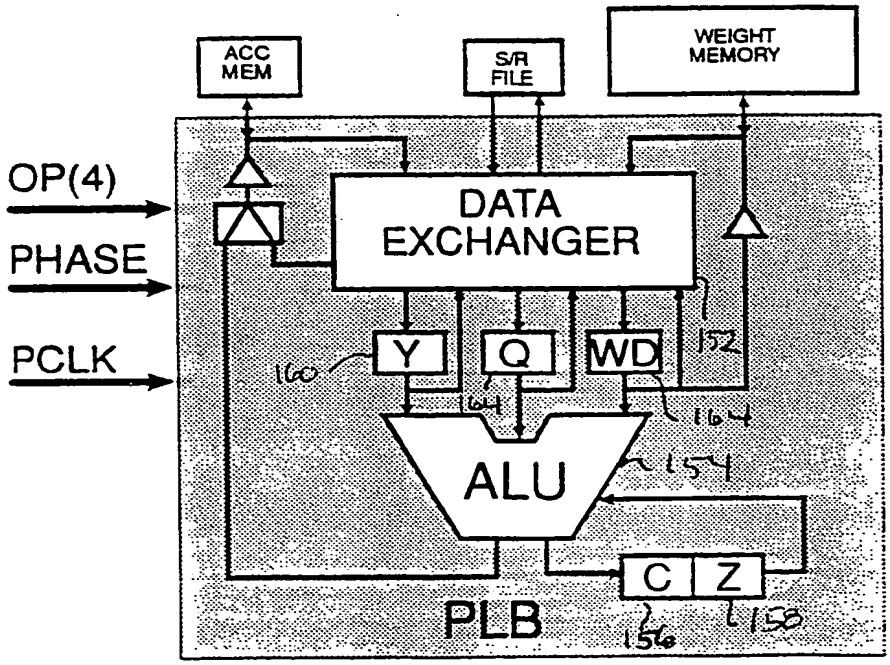

ACC MEM
S/R FILE
WEIGHT MEMORY
OP(4)
PHASE
PCLK
DATA EXCHANGER
Y
Q
WD
160
162
164
ALU
154
C
Z
PLB
156
158

FIG. 15

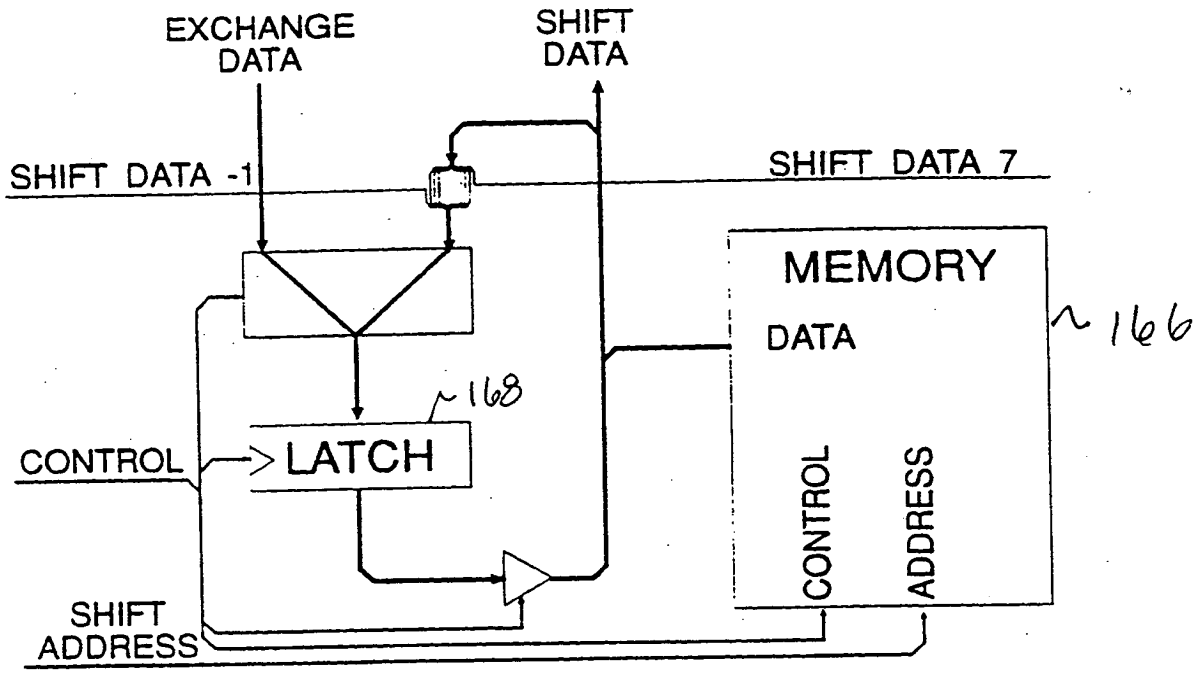


FIG. 16

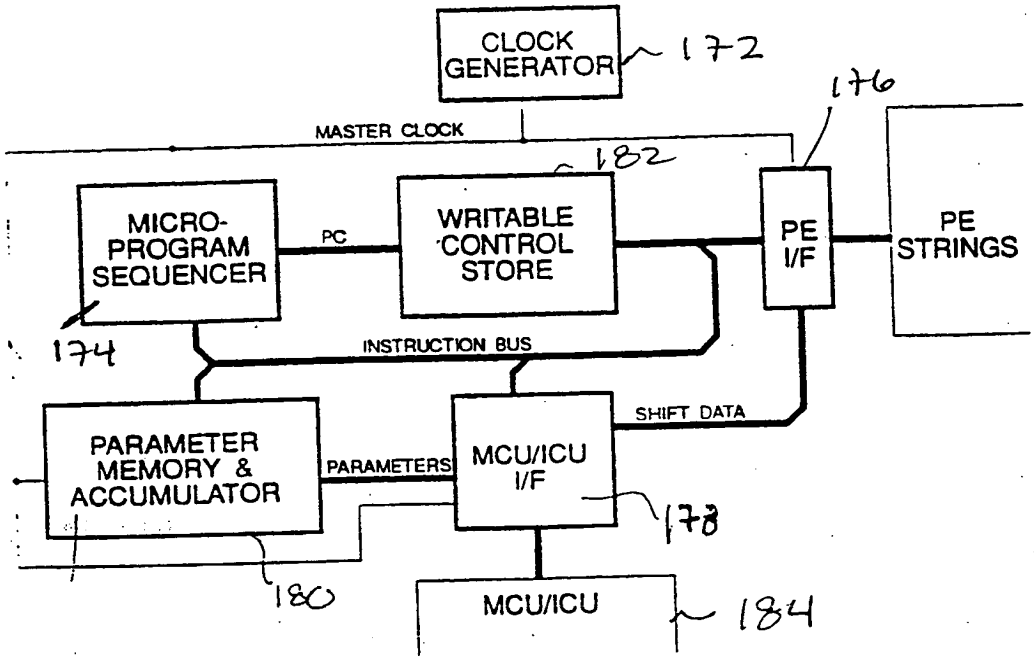


FIG. 17

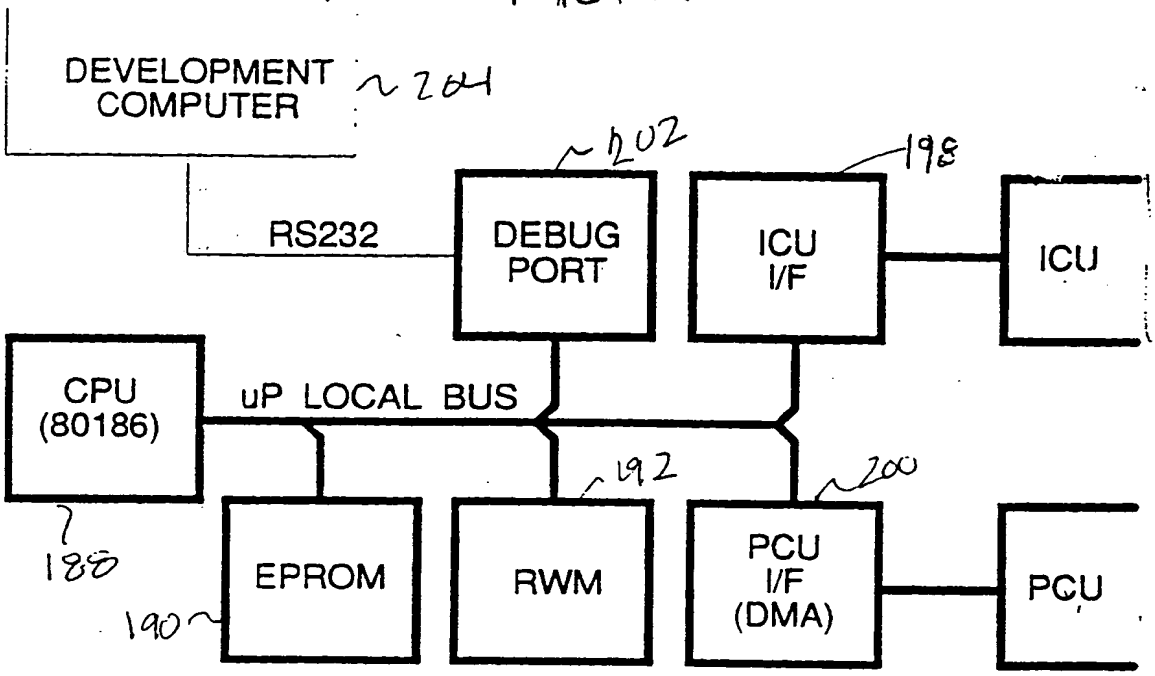

DEVELOPMENT COMPUTER
204
RS232
DEBUG PORT
202
ICU I/F
198
ICU
CPU (80186)
188
uP LOCAL BUS
EPROM
190
RWM
192
PCU I/F (DMA)
200
PCU

FIG. 18

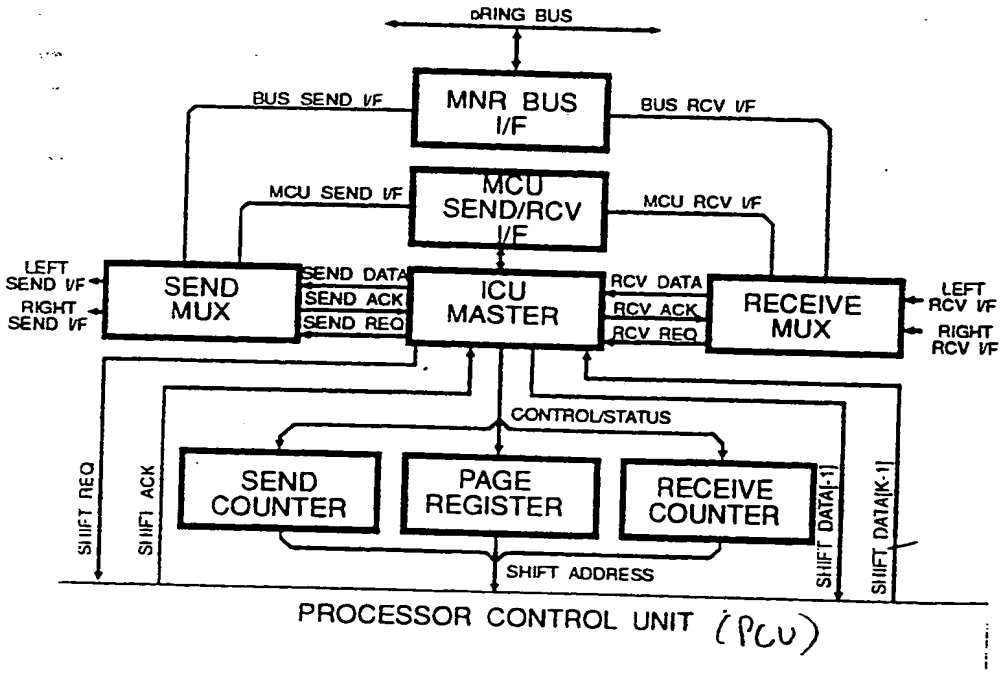

pRING BUS
MNR BUS I/F
BUS SEND I/F
BUS RCV I/F
MCU SEND/RCV I/F
MCU SEND I/F
MCU RCV I/F
LEFT SEND I/F
RIGHT SEND I/F
SEND MUX
SEND DATA
SEND ACK
SEND REQ
ICU MASTER
RCV DATA
RCV ACK
RCV REQ
RECEIVE MUX
LEFT RCV I/F
RIGHT RCV I/F
CONTROL/STATUS
SEND COUNTER
PAGE REGISTER
RECEIVE COUNTER
SHIFT REQ
SHIFT ACK
SHIFT DATA[1-1]
SHIFT DATA[K-1]
SHIFT ADDRESS
PROCESSOR CONTROL UNIT (PCU)

FIG. 19

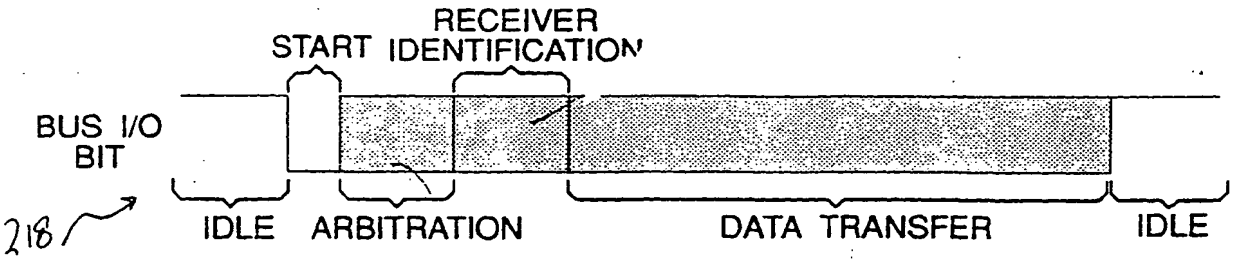


FIG. 20

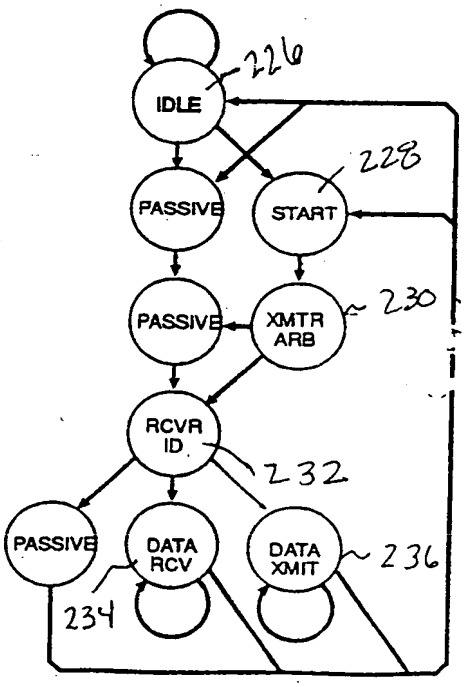


FIG. 21

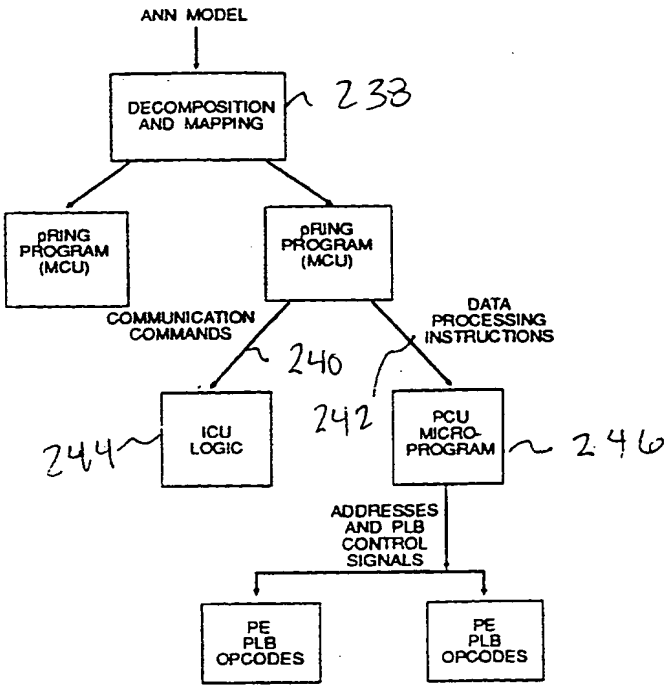
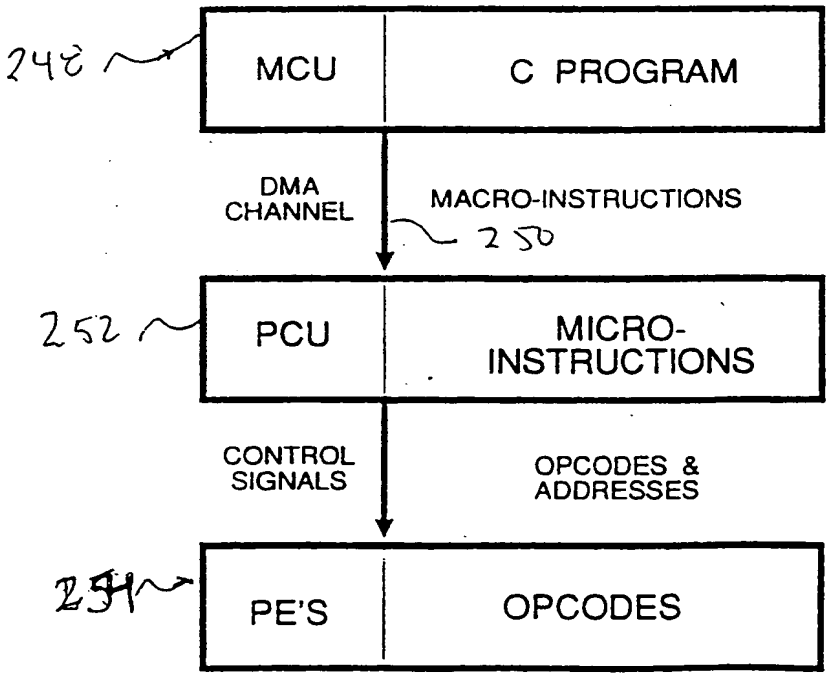

ANN MODEL
DECOMPOSITION AND MAPPING
233
pRING PROGRAM (MCU)
pRING PROGRAM (MCU)
COMMUNICATION COMMANDS
240
DATA PROCESSING INSTRUCTIONS
242
244
ICU LOGIC
PCU MICRO-PROGRAM
246
ADDRESSES AND PLB CONTROL SIGNALS
PE PLB OPCODES
PE PLB OPCODES

FIG. 22


248
MCU
C PROGRAM
DMA CHANNEL
MACRO-INSTRUCTIONS
250
252
PCU
MICRO-INSTRUCTIONS
CONTROL SIGNALS
OPCODES & ADDRESSES
254
PE'S
OPCODES

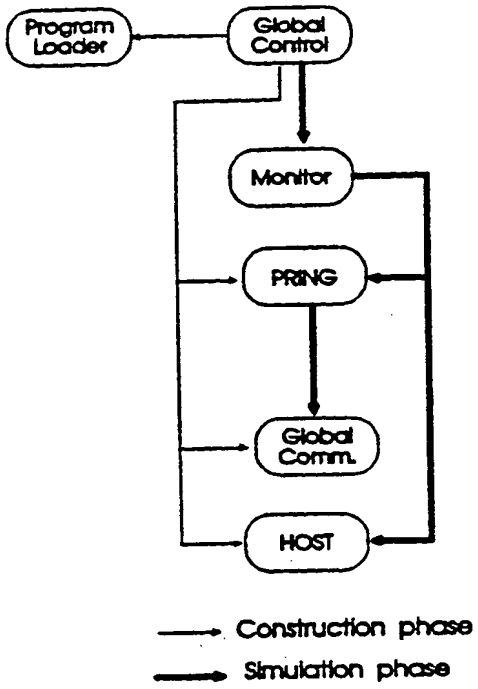


Figure 23

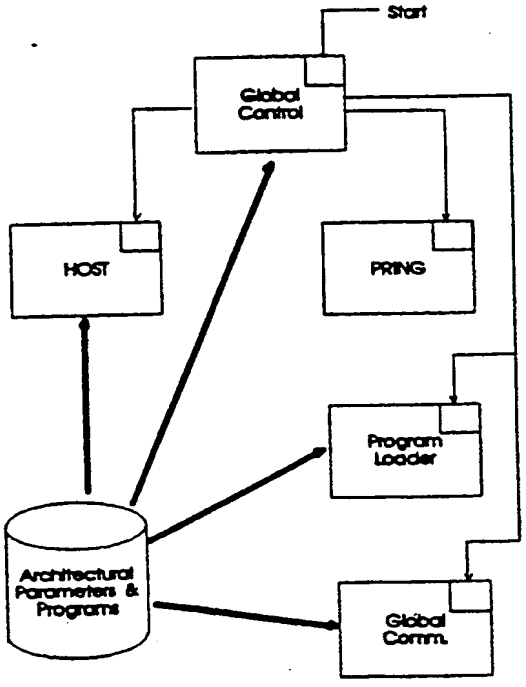


Figure 24

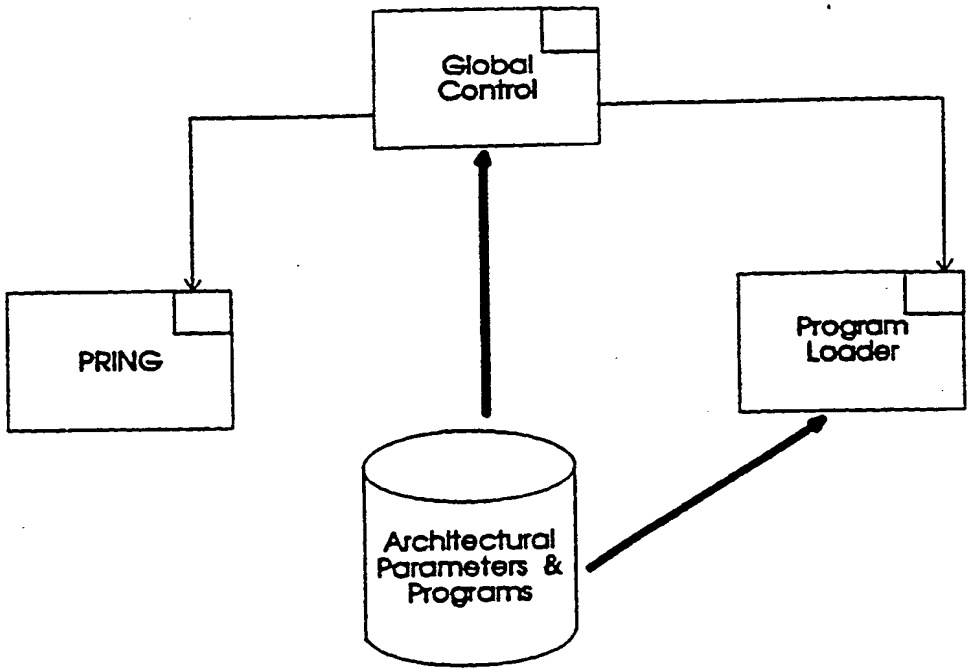


Figure 26

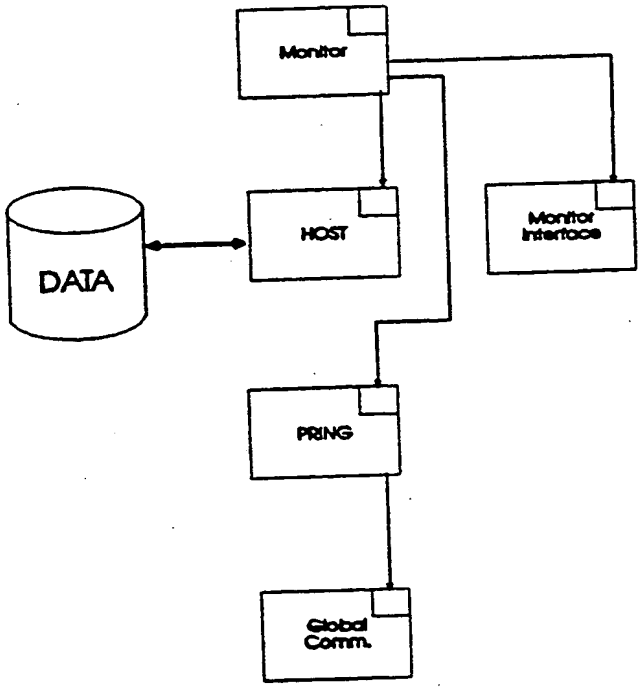


Figure 25

FIG. 27

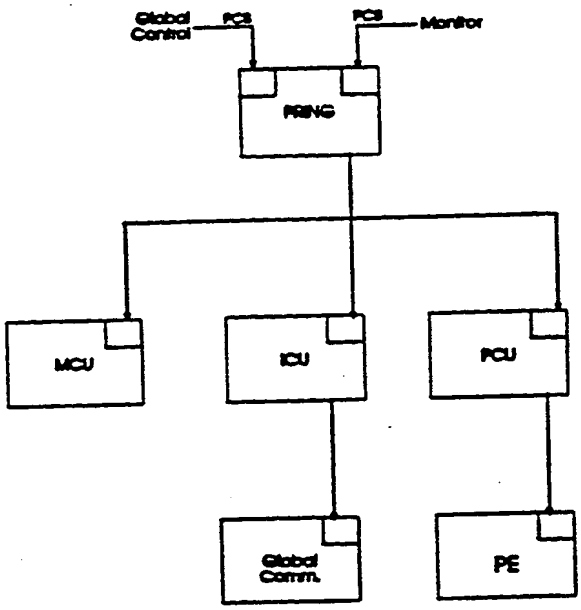

Global Control
PCS
PCS
Monitor
PRING
MCU
ICU
PCU
Global Comm.
PE

FIG. 28

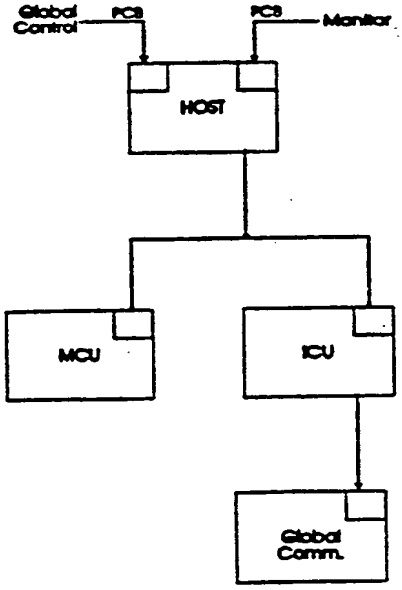

Global Control
PCS
PCS
Monitor
HOST
MCU
ICU
Global Comm.

FIG. 29

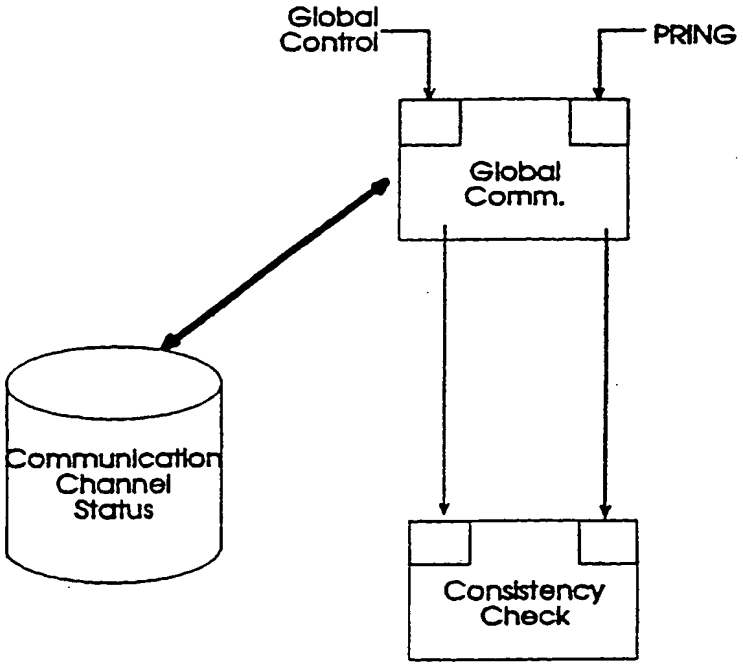
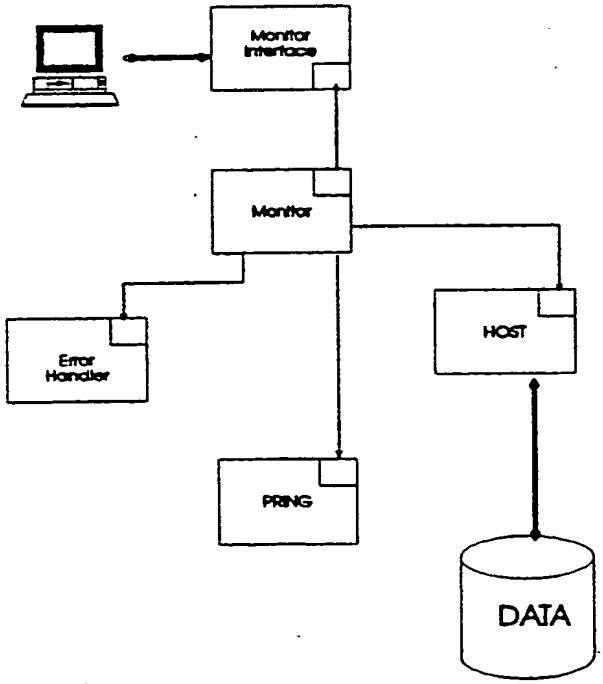

Global Control
PRING
Global Comm.
Communication Channel Status
Consistency Check

FIG. 30


Monitor Interface
Monitor
Error Handler
HOST
PRING
DATA

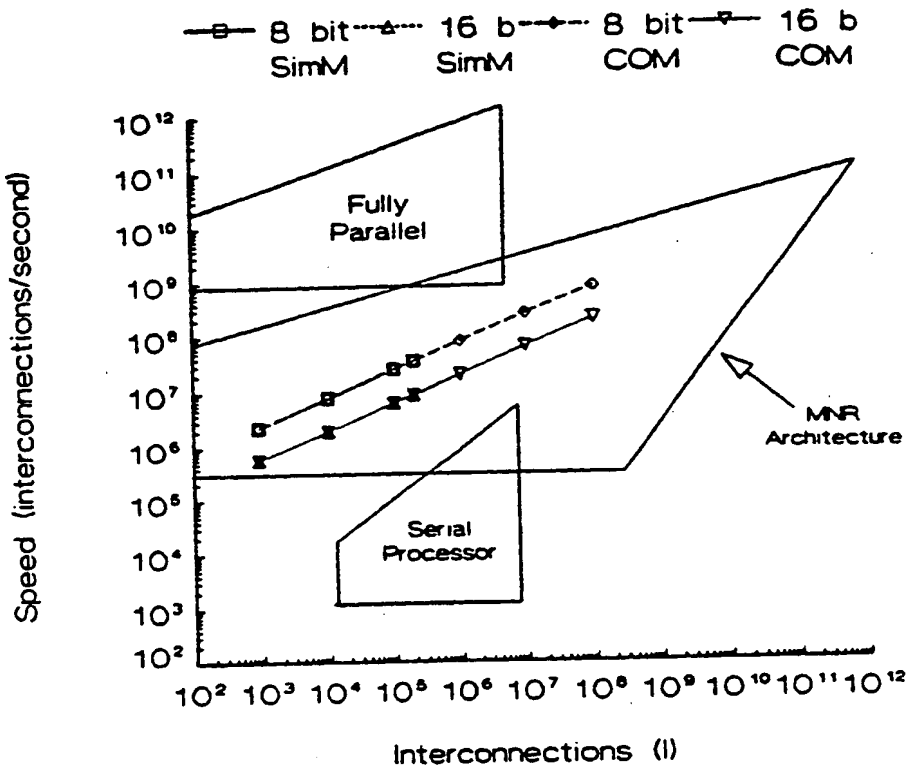


FIG. 3I

Performance of The MNR Architecture On the DARPA Map

CONDITIONS:

1. N/K = 1

2. M = 1

3. T_{mac}=200ns

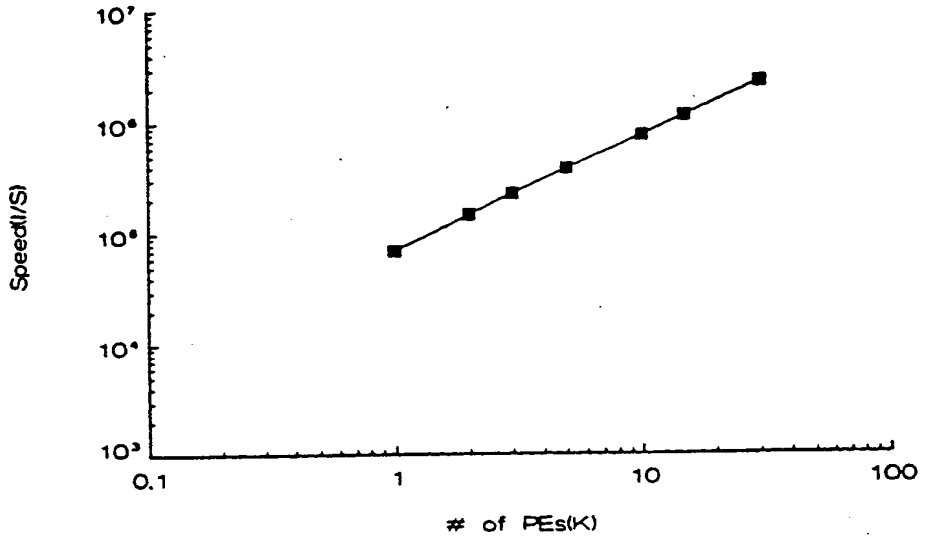


FIG. 32

Speed Vs. Number of PEs

CONDITIONS:

1. P = 8 bits

2. M = 1

3. N = 30 (I = 900)

4. T_{mac}=200ns

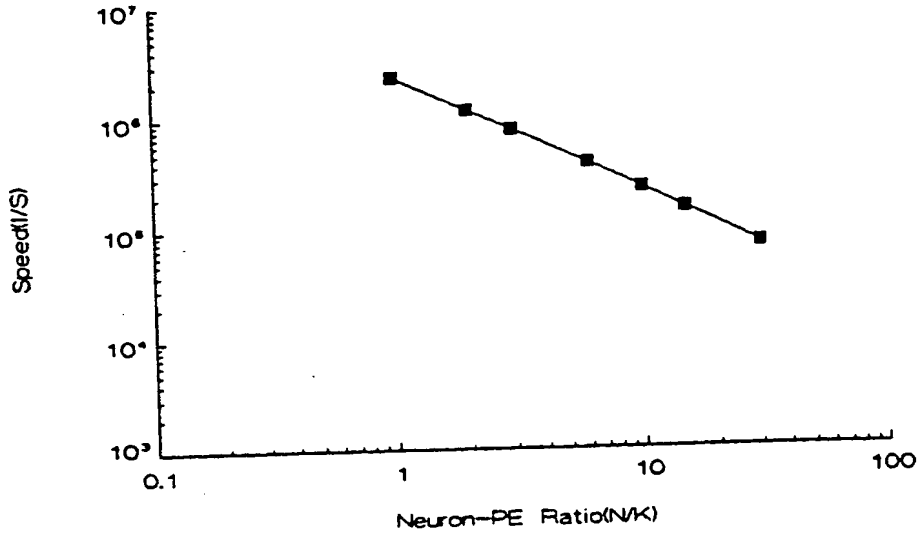


FIG. 33

Speed Vs. Neuron PE Ratio

CONDITIONS:

1. $P = 8$ *bits*
2. $M = 1$
3. $N = 30$ $(I = 900)$
4. $T_{mac} = 200ns$

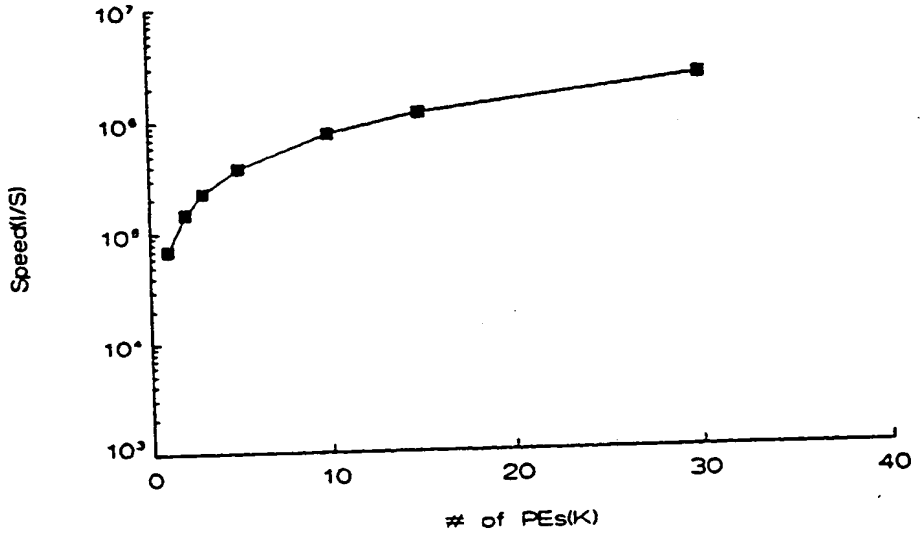


FIG. 34

Speed Vs. Number of PEs(linear)

CONDITIONS:

1. P = 8 bits

2. M = 1

3. N = 30 (I = 900)

4. T_{mac} = 200ns

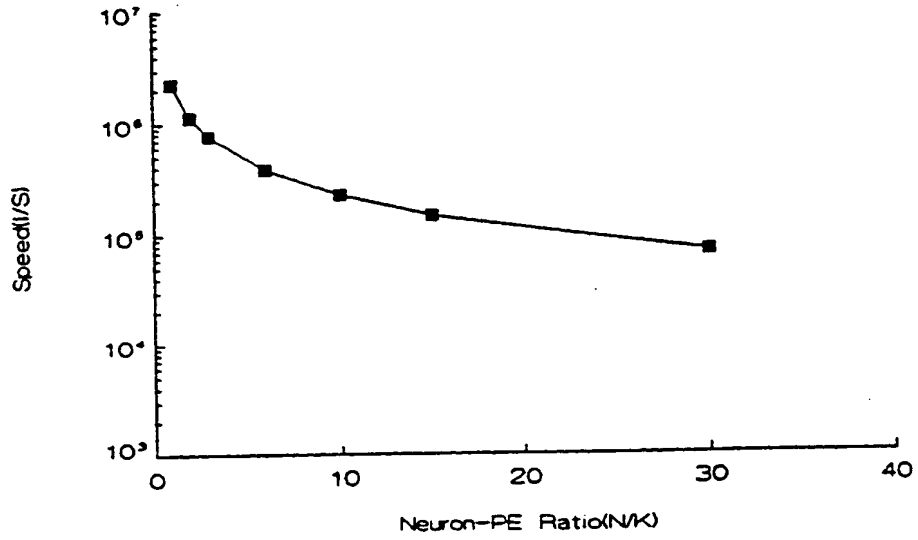


FIG. 35

Speed Vs. Neuron-PE Ratio(linear)

CONDITIONS:

1. P = 8 bits

2. M = 1

3. N = 30 (I = 900)

4. T_{mac}=200ns

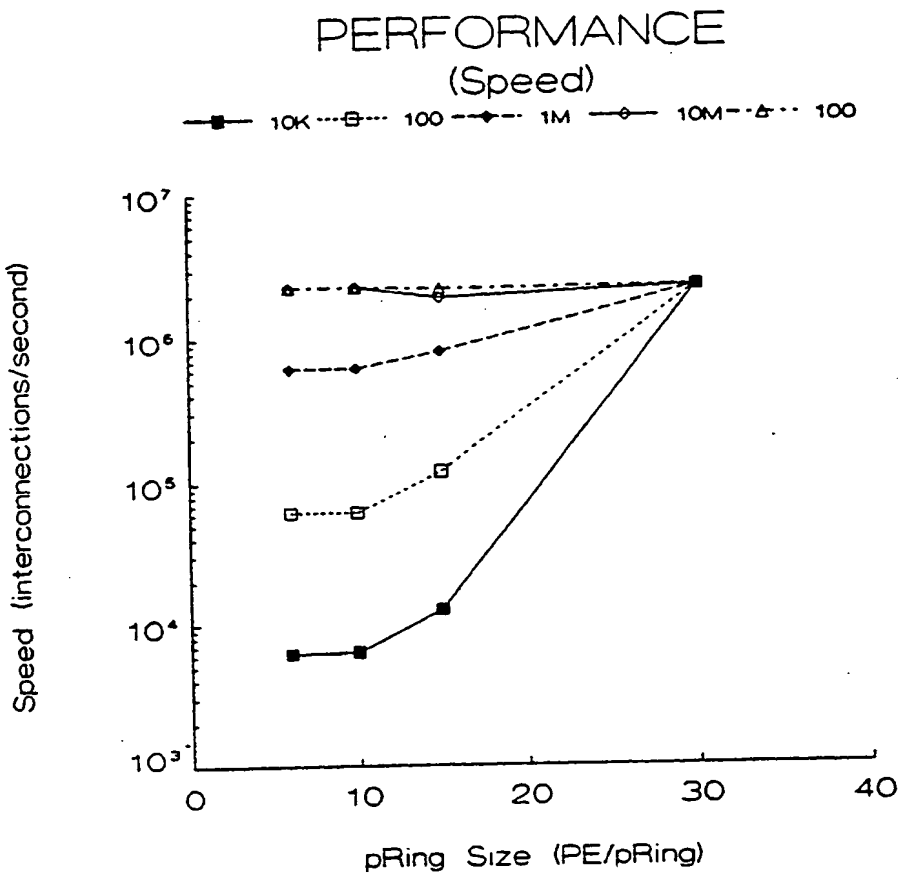


FIG. 36

Speed Vs. pRing Size (Varying B)

CONDITIONS:

1. $P = 8$ bits
2. $K = 30$
3. $N = 30$ ($I = 900$)
4. $T_{mac} = 200ns$

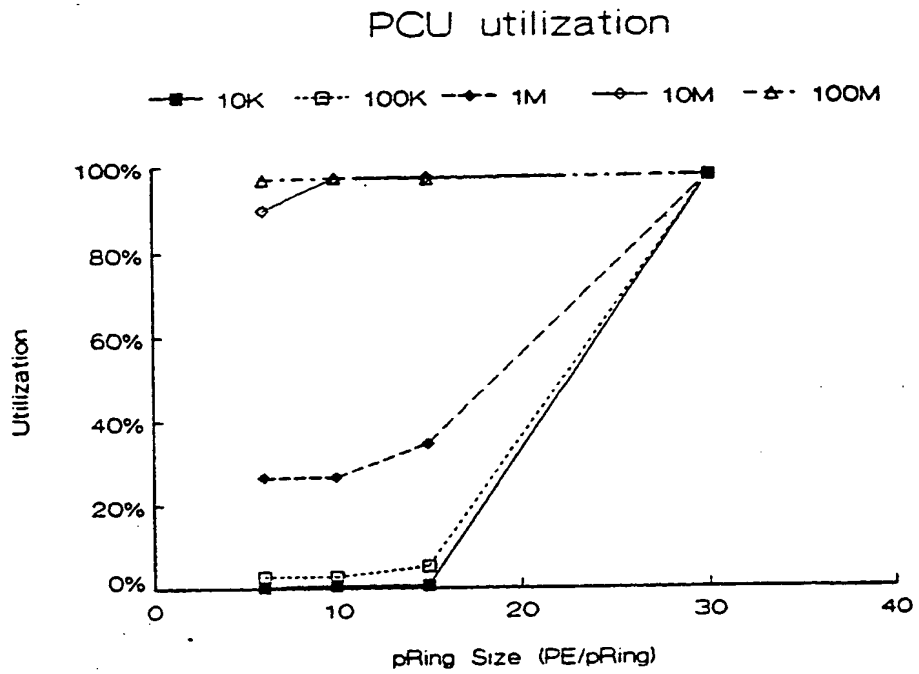


FIG. 37

PCU Utilization Vs. pRing Size (Varying B)

CONDITIONS:

1. P = 8 bits

2. K = 30

3. N = 30 (I = 900)

4. T_{mac} = 200ns

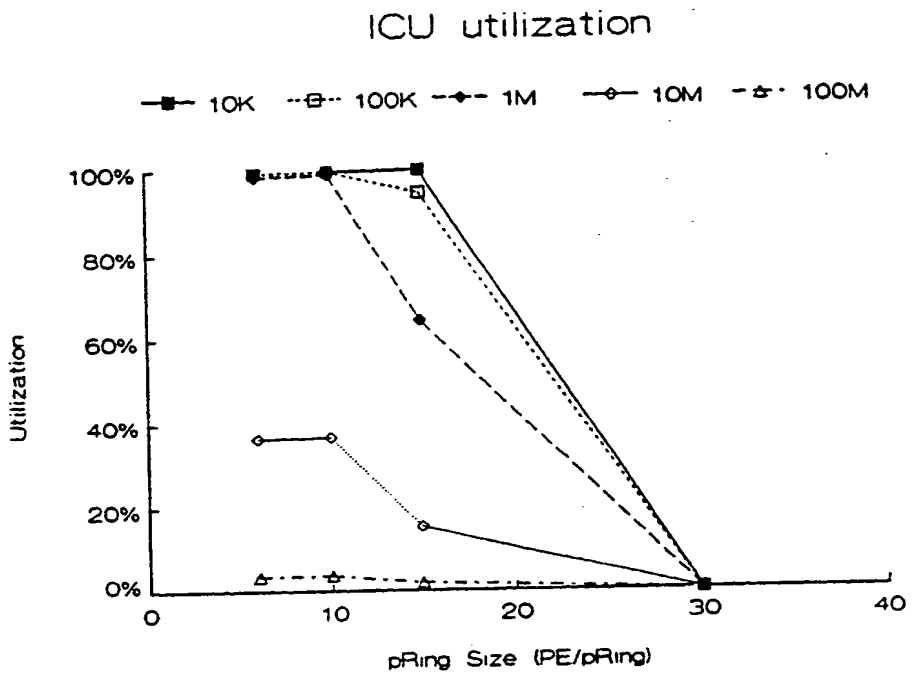


FIG. 3B

ICU Utilization Vs. pRing Size (Varying B)

CONDITIONS:

1. *P = 8 bits*
2. *K = 30*
3. *N = 30 (I = 900)*
4. *T_{mac} = 200ns*

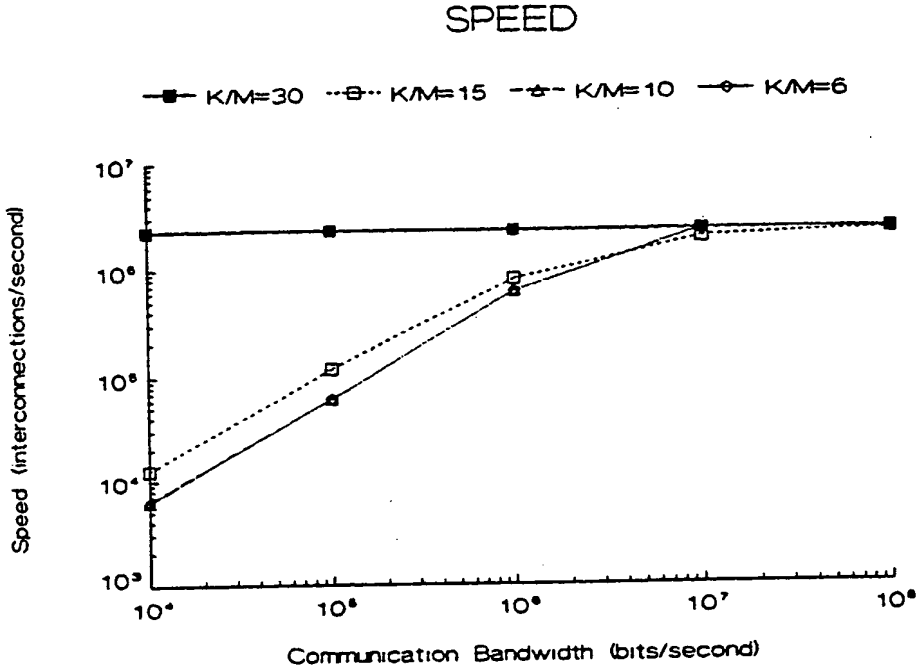


FIG. 39

Speed Vs. Communication Bandwidth (Varying M)

CONDITIONS:

1. P = 8 bits

2. K = 30

3. N = 30 (I = 900)

4. T_{mac} = 200ns

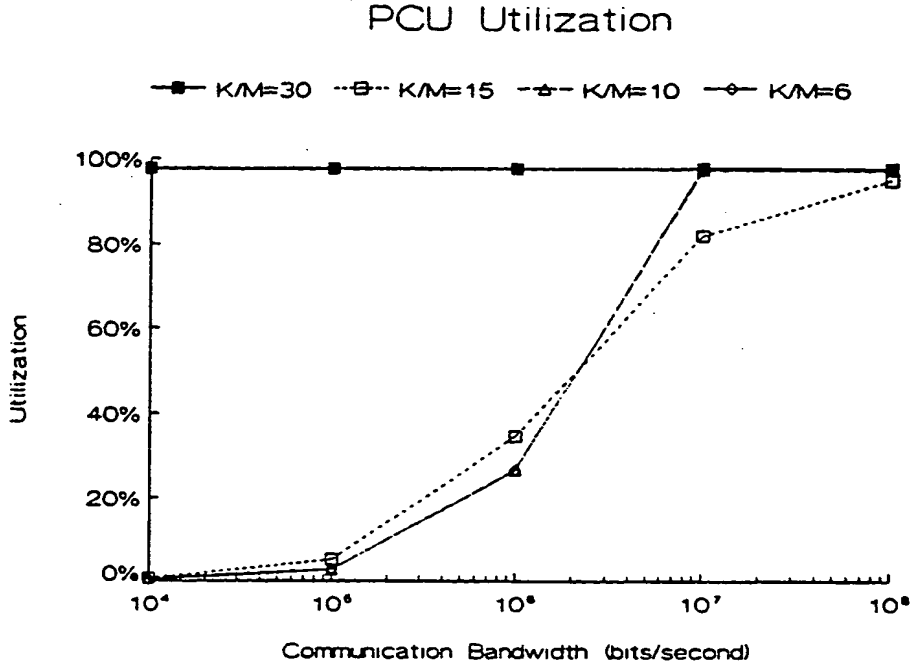


FIG. 40

PCU Utilization Vs. Communication Bandwidth (Varying M)

CONDITIONS:

1. P = 8 bits

2. K = 30

3. N = 30 (I = 900)

4. T_{mac} = 200ns

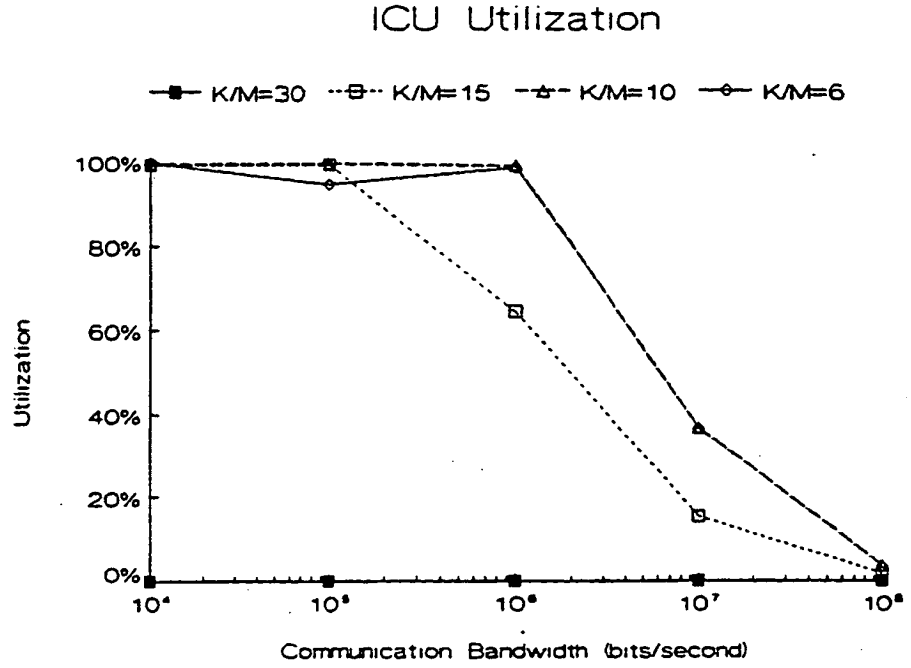


FIG. 41

ICU Utilization Vs. Communication Bandwidth (Varying M)

CONDITIONS:

1. $P = 8$ bits
2. $K = 30$
3. $N = 30$ ($I = 900$)
4. $T_{mac} = 200ns$

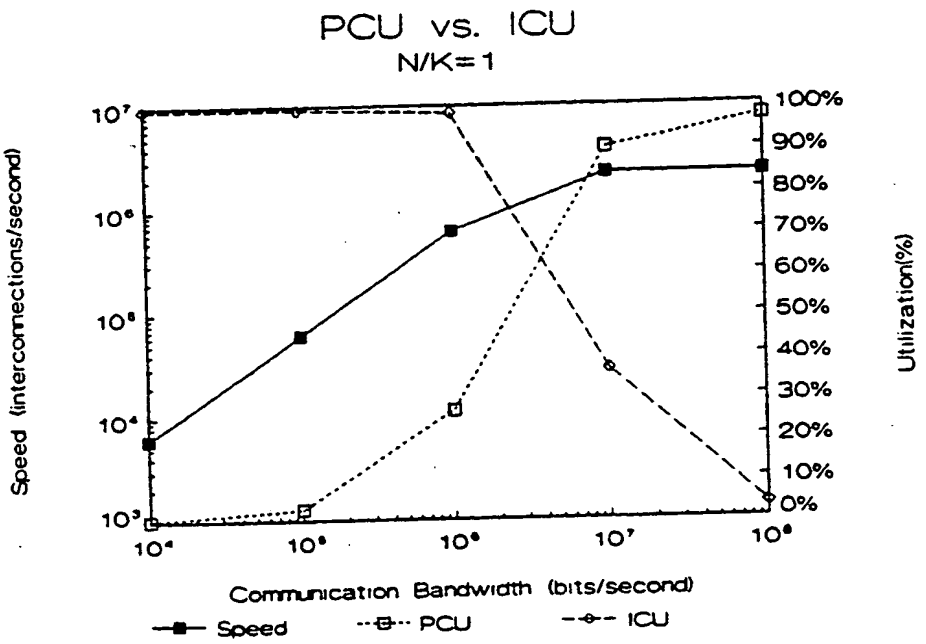


FIG. 4A

Device Utilization Comparison (N/K=1)

CONDITIONS:
1. P = 8 bits
2. M = 5
3. K = 30
4. N = 30 (I = 900)
5. T_{mac} = 200ns

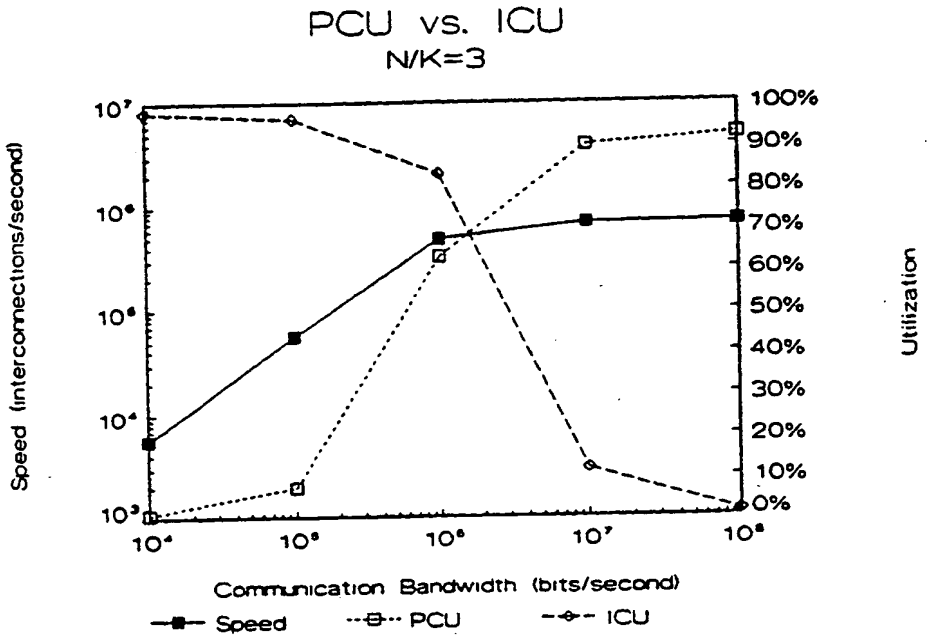


FIG. 43

Devices Utilization Comparison (N/K=3)

CONDITIONS:
1. P = 8 bits
2. M = 5
3. K = 10
4. N = 30 (I = 900)
5. T_{mac} = 200ns

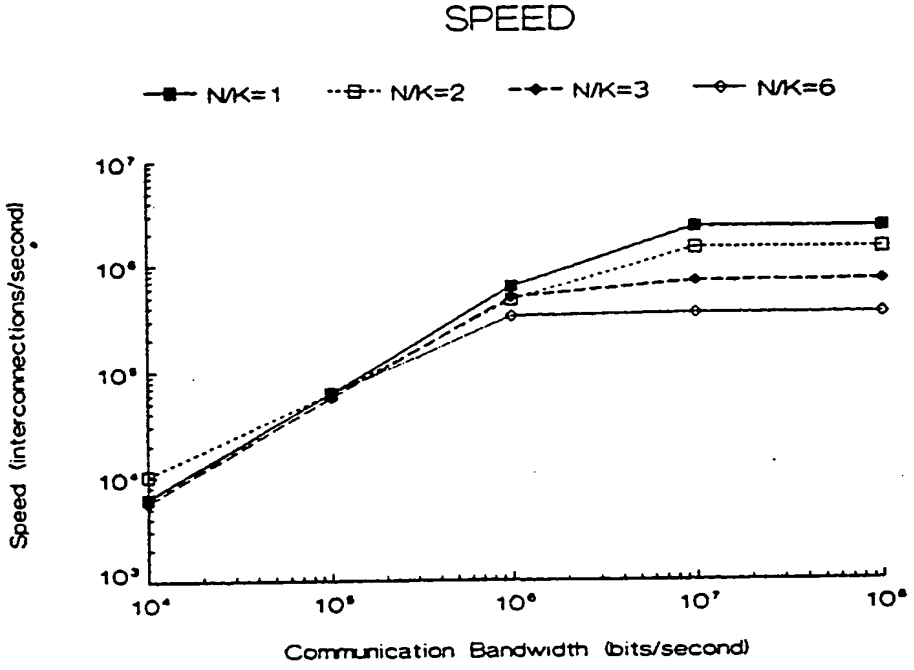


FIG. 44

Speed Vs. Communication Bandwidth (Varying N/K)

CONDITIONS:

1. P = 8 bits

2. M = 5

3. N = 30 (I = 900)

4. T_{mac} = 200ns

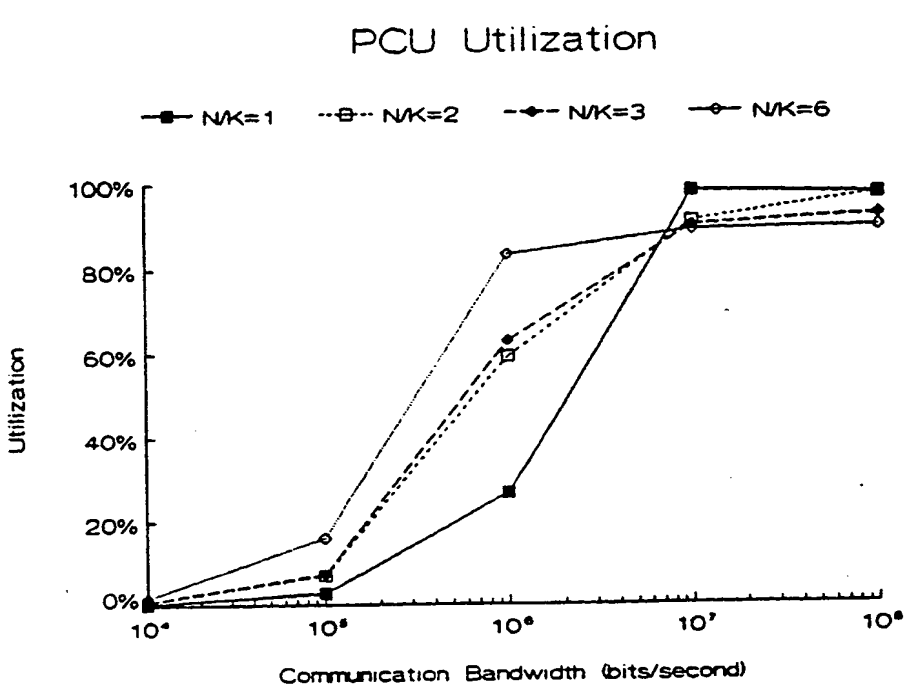


FIG. 45

PCU Utilization Vs. Communication Bandwidth (Varying N/K)

CONDITIONS:

1. P = 8 bits

2. M = 5

3. N = 30 (I = 900)

4. T_{mac} = 200ns

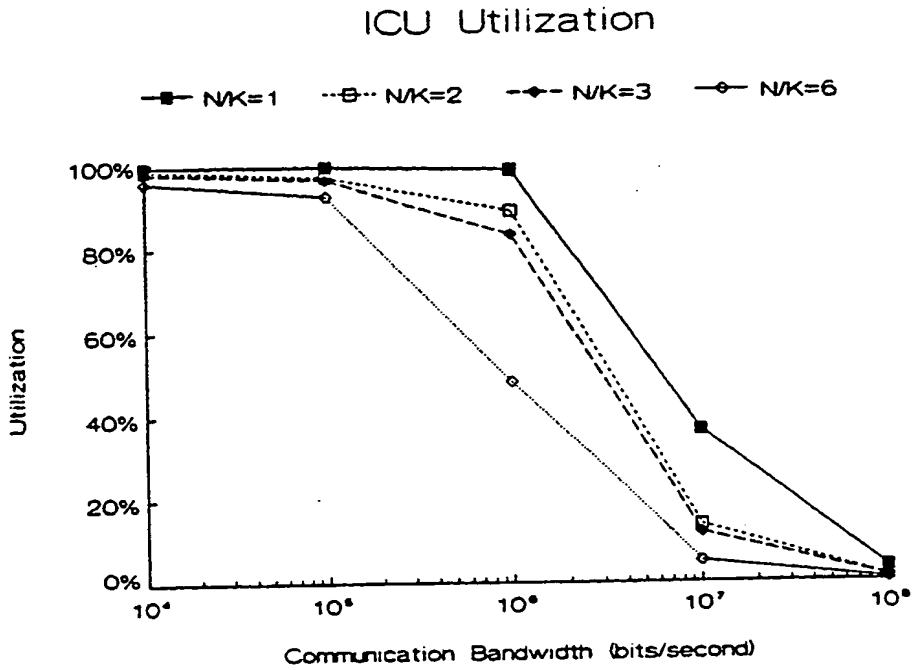


FIG. 4G

ICU Utilization Vs. Communication Bandwidth (Varying N/K)

CONDITIONS:

1. $P = 8$ bits
2. $M = 5$
3. $N = 30$ ($I = 900$)
4. $T_{mac} = 200ns$

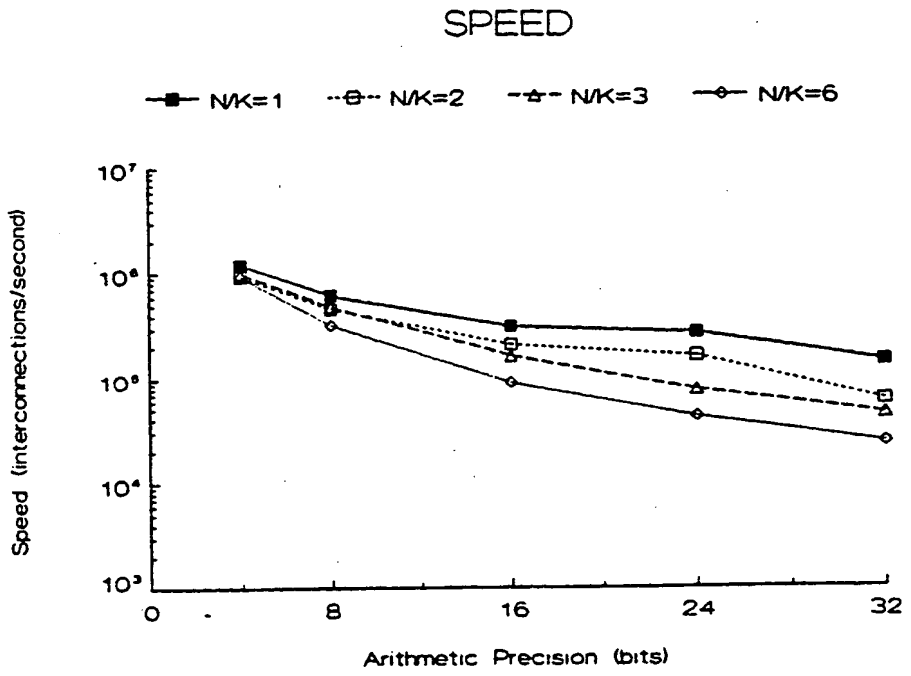


FIG. 47

Speed Vs. Precision (Varying K)

1. B = 1 Mbits/second
2. M = 5
3. N = 30 (I = 900)
4. T_{mac} = 200ns

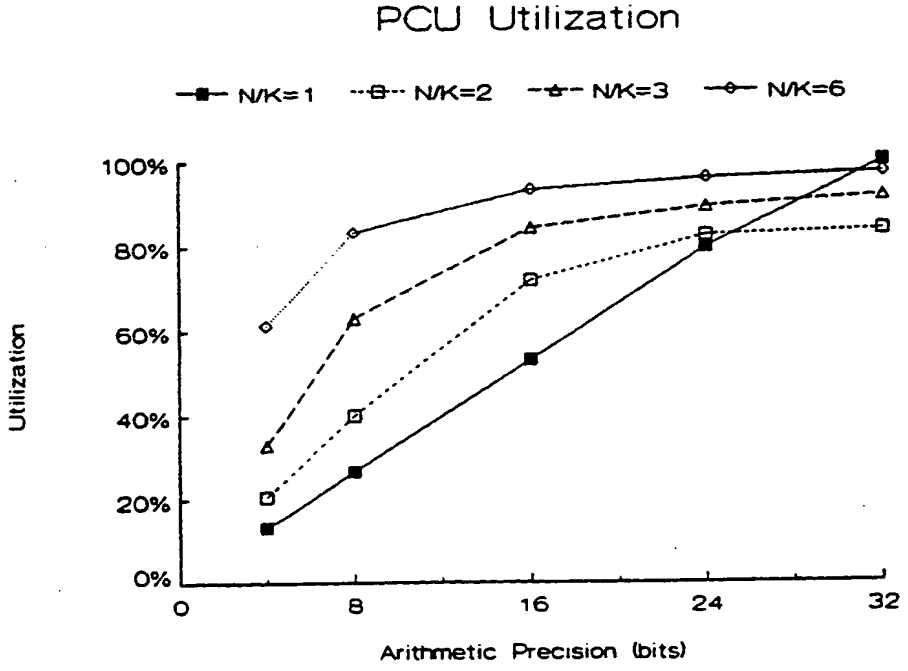


FIG. 48

PCU Utilization Vs. Precision (Varying K)

1. *B = 1 Mbits/second*
2. *M = 5*
3. *N = 30 (I = 900)*
4. *T_{mac} = 200ns*

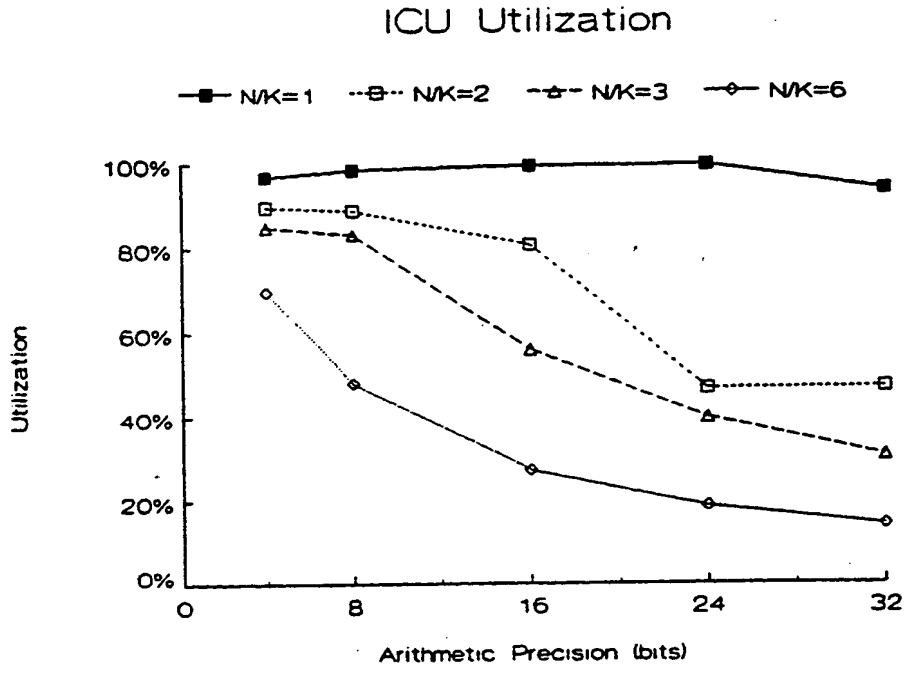


FIG. 49

ICU Utilization Vs. Precision (Varying K)

1. B = 1 Mbits/second

2. M = 5

3. N = 30 (I = 900)

4. T_{mac} = 200ns

FIG. 50

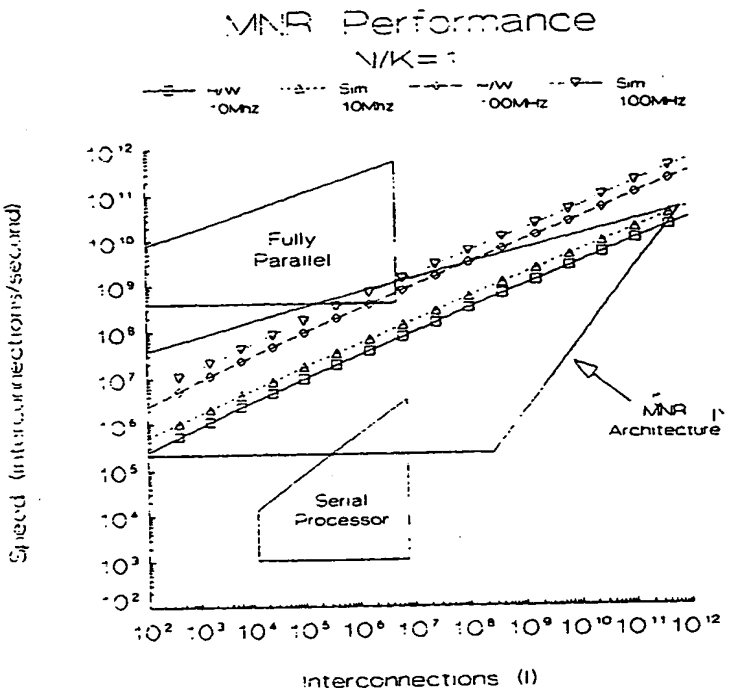

MNR Performance
N/K=1
Sim 10Mhz
Sim 100Mhz
Fully Parallel
MNR Architecture
Serial Processor
Speed (interconnections/second)
Interconnections (I)

FIG. 51

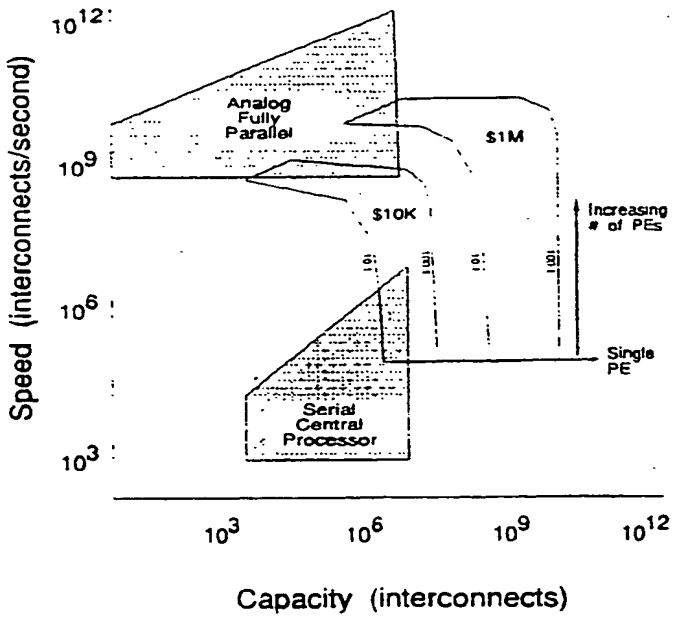

Analog Fully Parallel
$1M
$10K
Increasing # of PEs
Single PE
Serial Central Processor
Speed (interconnects/second)
Capacity (interconnects)

FIG. 52

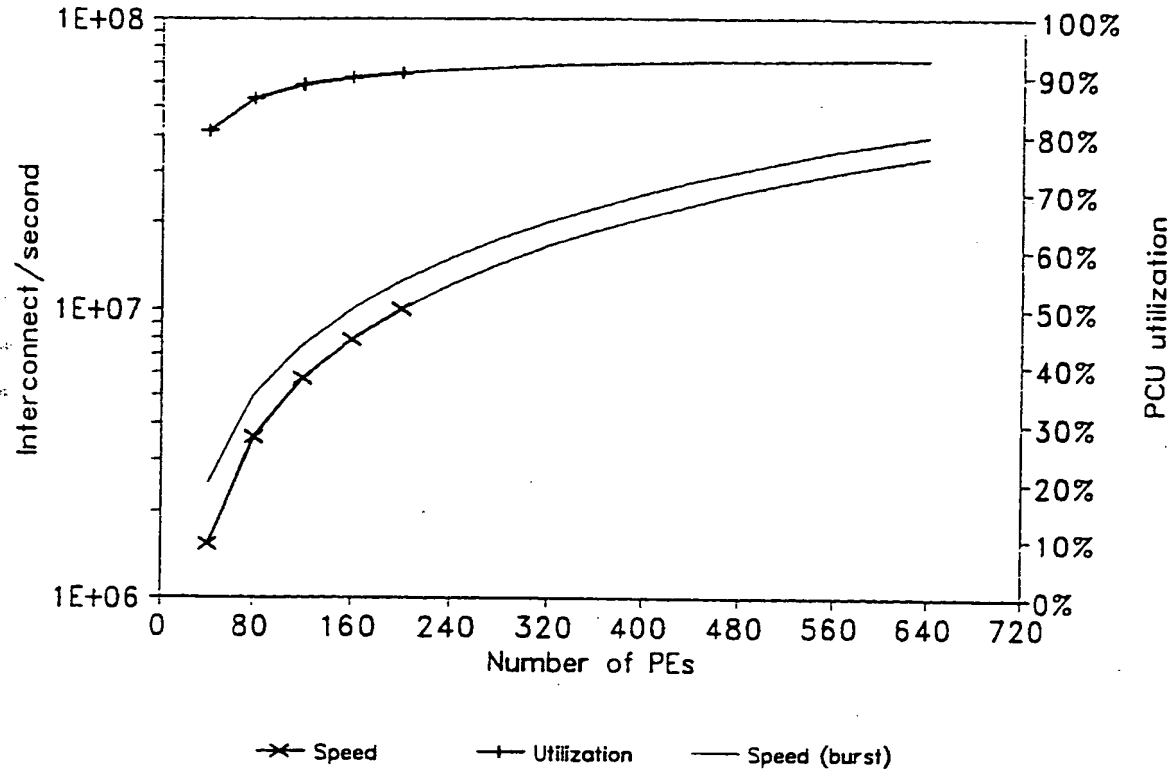


FIG. 53

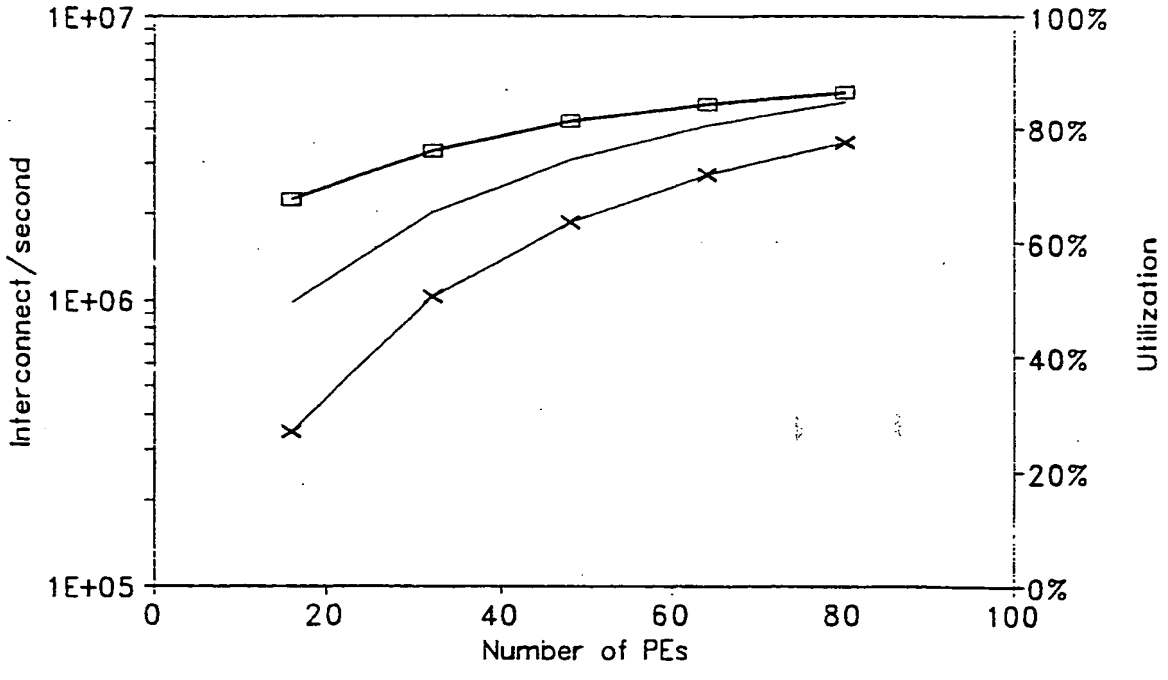


FIG 54

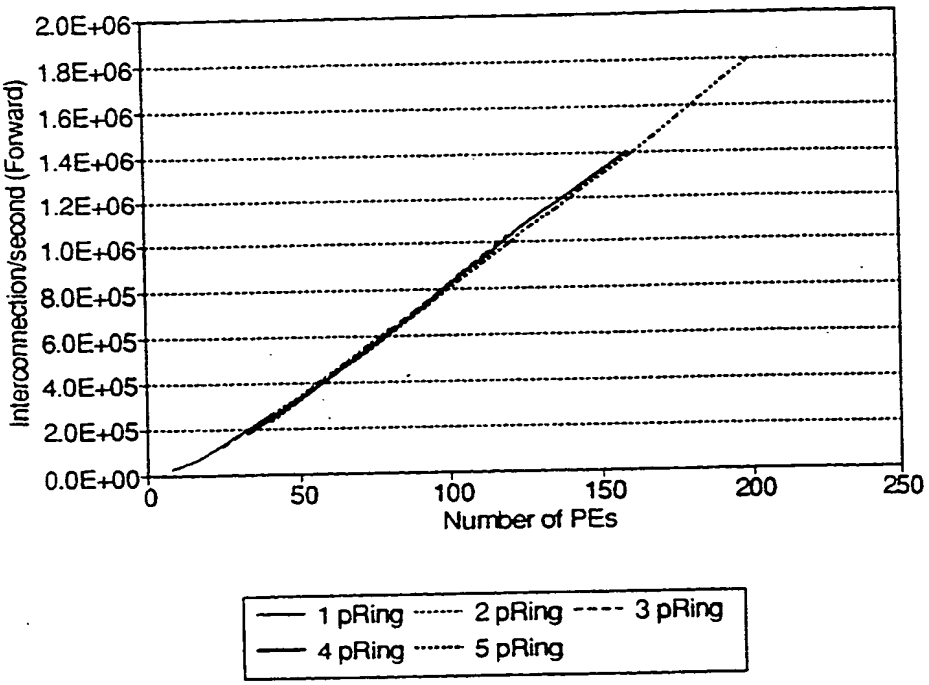


FIG. 55

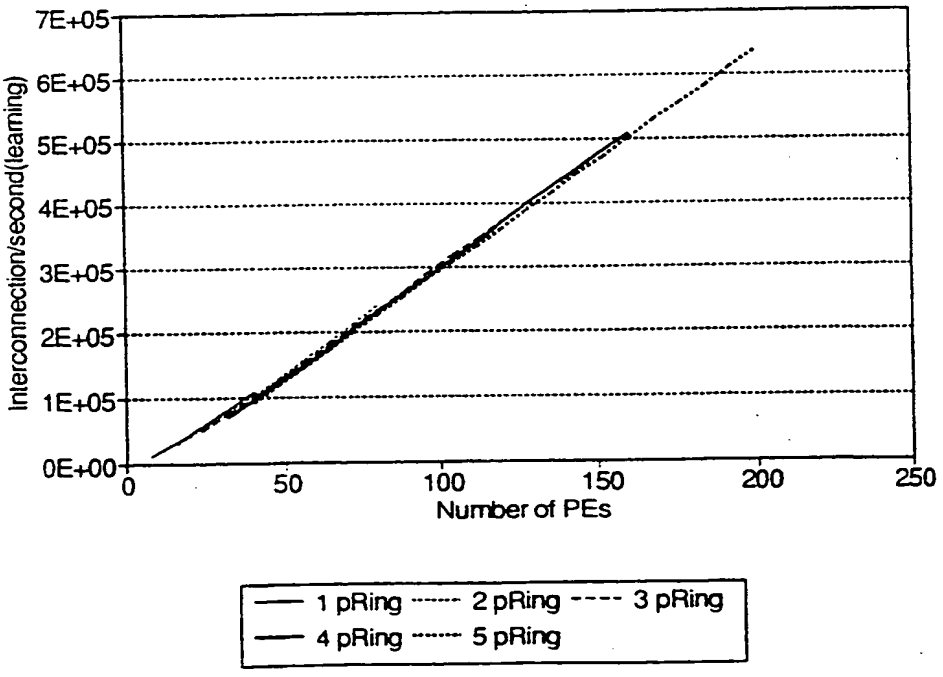
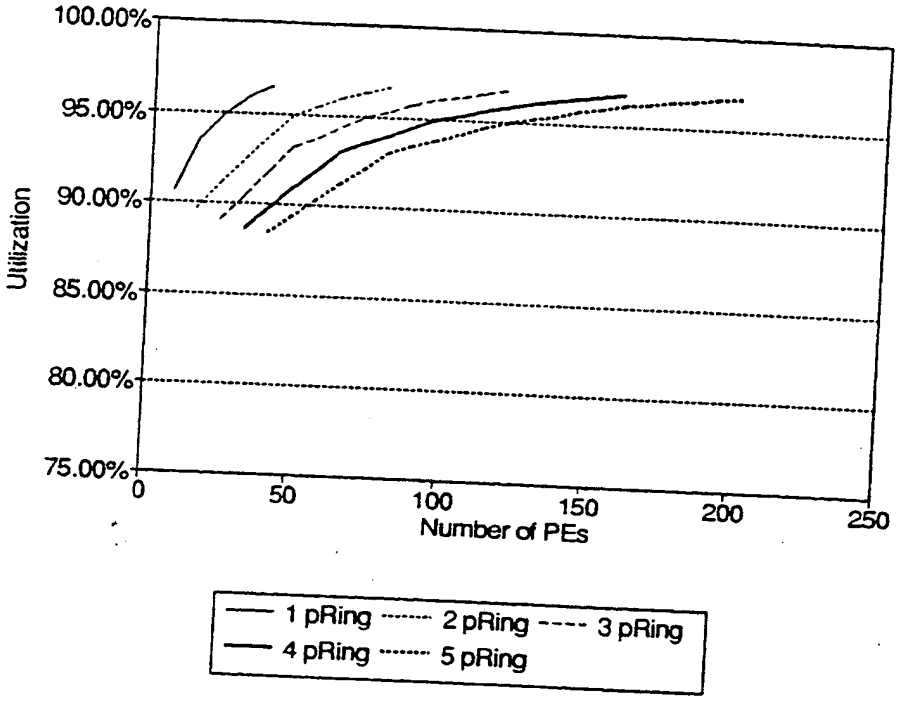


FIG. 5B


100.00%
95.00%
90.00%
85.00%
80.00%
75.00%
Utilization
0
50
100
150
200
250
Number of PEs
1 pRing
2 pRing
3 pRing
4 pRing
5 pRing